Generación de lenguaje natural a partir de clases de prueba del test template framework

Tesina de Grado
Licenciatura en Ciencias de la Computación

Julian De Tomasi

Director
Maximiliano Cristiá

Co-Director
Brian Plüss

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

XXXX 2015

Índice general

Capítulo 1

Introducción

1.1. Motivación y objetivo general

El testing basado en modelos (abreviado **MBT** del inglés *"model based testing"*) es una de las técnicas de testing más prometedoras para la verificación de software crítico. Estas metodologías comienzan con un modelo formal o especificación del software, de la cual son generados los casos de prueba.

Un caso particular del testing basado en modelos es el *Test Template Framework* (**TTF**), descrito por Stocks y Carrington [SC96], el cual utiliza como modelo de entrada especificaciones formales escritas en notación *Z* y establece cómo generar casos de prueba para cada operación incluida en el modelo. Esta técnica genera descripciones lógicas, también en lenguaje Z, de los casos de prueba. El TTF propone en primera instancia obtener casos de prueba abstractos a partir de una especificación, cada uno de estos probará una alternativa funcional distinta del sistema a testear. Cada una de estas alternativas será expresada como un esquema Z llamado *clase de prueba* y luego, a partir de las mismas, se generarán los casos de *prueba concretos*.

En la figura 1.3 podemos observar, a modo de ejemplo, la clase de prueba *Update_SP_4* junto al caso de prueba *Update_SP_4_TCASE* generados para testear la operación *Update* que modela la actualización de una tabla de símbolos (en el capítulo 2.1 presentaremos la especificación completa).

$$
\begin{array}{|l}
\hline
Update\_SP\_4 \\
st : SYM \pfun VAL \\
s? : SYM \\
v? : VAL \\
\hline
\mathrm{dom}\, st = \mathrm{dom}\{s? \mapsto v?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Update\_SP\_4\_TCASE \\
Update\_SP\_4 \\
\hline
st = \{(sym0, val0)\} \\
s? = sym0 \\
v? = val0 \\
\hline
\end{array}
$$

Figura 1.1: Clase y caso de prueba para operación Update.

Por otro lado, es una practica común cuando se especifica formalmente un sistema, incluir *designaciones* [Jac95] entre elementos de la especificación (operaciones, esquemas de estado, variables, constantes, etc.) y elementos que refieran al dominio de la aplicación. En la figura 1.2, podemos ver algunas de las designaciones que deberían acompañar la especificación de la tabla de símbolos antes mencionada.

Update	$\approx$ se intenta actualizar un símbolo en la tabla.
dom st	$\approx$ símbolos cargados en la tabla de símbolos.
$s?$	$\approx$ símbolo a actualizar.

Figura 1.2: Designaciones especificación SymbolTable.

El desarrollo de software crítico usualmente requiere de procesos independientes de validación y verificación. Estos procesos son llevados a cabo por expertos en el dominio de aplicación, quienes usualmente no poseen conocimientos técnicos, en nuestro caso en particular, necesitaríamos que la persona que realiza la verificación y validación sea capaz de leer Z para comprender lo que está siendo testeado. En estos casos, una descripción en lenguaje natural de cada caso de prueba debería acompañar a los mismos a fin de hacerlos accesibles para los expertos en el dominio. Por ejemplo, en la figura 1.3 mostramos una posible descripción para el caso de prueba del ejemplo anterior.

Update_SP_4: Se actualiza un símbolo en la tabla.

- Cuando:
  - El símbolo a actualizar es el único símbolo cargado en la tabla de símbolos.

Figura 1.3: Descripción en lenguaje natural para *Update_SP_4*.

Contar con una descripción en lenguaje natural como la previamente mencionada, sería de gran ayuda para que la persona a cargo de la validación y verificación comprenda lo que está siendo testeado.

En sistemas en los que hay una gran cantidad de casos de prueba, traducir manualmente los mismos podría introducir errores humanos, reduciendo la calidad de las descripciones además de incrementar el costo total del testing.

El objetivo general de este trabajo, será entonces, desarrollar una solución para la generación automática de descripciones de casos de prueba generados por el TTF, trabajando fundamentalmente con la información contenida en las clases de prueba y haciendo uso de las designaciones antes mencionadas a fin de lograr una solución independiente del dominio de aplicación y del número de operaciones a testear. Para esto, utilizaremos técnicas de *generación de lenguaje natural* (abreviado **NLG** del inglés *"natural language generation"*) que es el área del procesamiento de lenguaje natural que estudia la producción automática de textos en alguna lengua humana a partir de una representación computacional de la información. En particular, seguiremos la metodología más comúnmente aceptada para la construcción de sistemas de NLG, propuesta por Reiter y Dale [RD00].

Como resultado de este trabajo también se realizó la implementación de un prototipo, desarrollado en Java e integrado a *Fastest*[1] (una implementación del TTF desarrollada por Crstiá y Monetti [CM09] capaz de generar casos de prueba a partir de una especificación Z) permitiendo la generación de descripciones de casos de prueba interactivamente desde la herramienta.

1.2. Antecedentes

Se han hecho variados esfuerzos para producir versiones en lenguaje natural de especificaciones formales. Punshon [PTSF97] usó un caso de estudio para presentar el sistema REVIEW [SSTP94]. REVIEW parafraseaba automáticamente especificaciones desarrolladas con Metaview [STM88], un metasistema que facilita la construcción de entornos CASE para soportar tareas de especificación de software. Coscoy [Cos97] desarrolló un mecanismo basado en la extracción de programas, para generar explicaciones de pruebas formales en el cálculo de construcciones inductivas, implementado en Coq [BC10]. Lavoie [LRR97] presentó MODEX, una herramienta que genera descripciones

[1] http://www.flowgate.net/tools/

personalizables de las relaciones entre clases en modelos orientados a objetos especificados en el estándar ODL [CBB+97]. Bertani [BCCM99] describió un enfoque para la traducción de especificaciones formales escritas en una extensión de TRIO [GMM90] en lenguaje natural controlado, transformando arboles sintácticos de TRIO en arboles sintácticos del lenguaje controlado.

Cristiá y Plüss [CP10] utilizaron métodos de generación de lenguaje natural basado en templates para la traducción de casos de prueba generados a partir de una especificación Z para un estándar aeroespacial. El trabajo anterior presenta una solución ad-hoc basada en templates, donde los templates utilizados son dependientes del dominio de aplicación y cantidad de operaciones.

Basado en el trabajo antes mencionado intentaremos desarrollar una solución independiente del dominio de aplicación y del número de operaciones del sistema. Para esto, trabajaremos fundamentalmente sobre las clases de prueba (que nos permitirán generar mejores descripciones), utilizando también la información contenida en las designaciones para lograr un resultado independiente del dominio.

1.3. Alcance del trabajo

Como mencionamos anteriormente, trabajaremos fundamentalmente a partir de clases de prueba generadas por el TTF escritas en notación Z. Para esto, tendremos en cuenta un subconjunto del total de los operadores presentes en Z, pudiéndose en el futuro, ampliar o extender el mismo. Creemos que el conjunto de operadores escogidos es lo suficientemente abarcativo, permitiéndonos trabajar con una gran variedad de especificaciones y casos de pruebas generados a partir de las mismas.

En la figura 1.4 podemos ver todos los operadores contemplados para este trabajo. Es decir, nuestro sistema de NLG deberá ser capaz de generar descripciones en lenguaje natural para todas las expresiones formadas a partir de estos operadores (incluyendo todas las expresiones que puedan surgir como combinaciones de los mismos).

1. $=$
2. $\neq$
3. $\in$
4. $\notin$
5. $\subset$
6. $\subseteq$
7. $\mapsto$
8. $\{a, ..., b\}$
9. $\cup$
10. $\cap$
11. f x (aplicación de función)
12. dom
13. ran
14. $+$
15. -
16. *
17. div
18. mód

Figura 1.4: Expresiones soportadas

Capítulo 2

Test Template Framework

El Test Template Framework (TTF) descrito por Stocks y Carrington [SC96] es un método de testing basado en modelos (MBT). Esta técnica permite efectuar un testing muy completo de un sistema del cual se posee una especificación Z [Spi92], utilizando la misma como entrada y estableciendo como generar casos de prueba para testear las distintas operaciones incluidas en el modelo.

La hipótesis fundamental detrás del testing basado en modelos es que, un programa es correcto si verifica su especificación, entonces la especificación resulta una excelente fuente para obtener casos de prueba. Una vez que los casos de prueba son derivados del modelo, estos son refinados al nivel del lenguaje de implementación y ejecutados. Luego la salida del programa es abstraída al nivel de la especificación y el modelo es usado nuevamente para verificar si el caso de prueba ha detectado un error.

A continuación introduciremos brevemente el TTF mediante un ejemplo, asumiendo que el lector se encuentra familiarizado con la notación Z.

2.1. Ejemplo: Symbol Table

Una tabla de símbolos es una estructura de datos utilizada por un compilador o interprete durante el proceso de traducción de un lenguaje de programación donde cada símbolo en el código del programa (variables, constantes, funciones, etc.) se asocia con información como la ubicación, tipo de datos, scope de variables, etc. En general en una tabla de símbolos se realizan dos operaciones: inserción y búsqueda. La primera para agregar un símbolo a la tabla y la segunda operación nos permitirá recuperar la información para un símbolo ya cargado en la tabla. La especificación Z de la Figura 2.1 modela la tabla y las dos operaciones que actúan sobre ella.

$[SYM, VAL]$

$REPORT ::= ok \mid symbolNotPresent$

ST

$st : SYM \pfun VAL$

Update

ΔST
$s? : SYM$
$v? : VAL$
$rep! : REPORT$

$st' = st \oplus \{s? \mapsto v?\}$
$rep! = ok$

LookUpOk

ΞST
$s? : SYM$
$v! : VAL$
$rep! : REPORT$

$s? \in \mathrm{dom}\, st$
$v! = st\ s?$
$rep! = ok$

LookUpE

ΞST
$s? : SYM$
$rep! : REPORT$

$s? \notin \mathrm{dom}\, st$
$rep! = symbolNotPresent$

$LookUp == LookUpOk \lor LookUpE$

Figura 2.1: Modelo Z para una tabla de símbolos.

Tanto los símbolos aceptados por el compilador/interprete, cómo la información de cada uno de estos serán representados mediante los siguientes tipos básicos, sin mayores detalles de los mismos:

$$[SYM, VAL]$$

Al abstraer toda la información asociada a un símbolo haciendo uso de los tipos básicos de Z podemos modelar la tabla de símbolos como una relación funcional entre símbolos y la información asociada a cada uno de ellos.

$$\begin{array}{|l} ST \\ \hline st : SYM \pfun VAL \\ \hline \end{array}$$

El esquema *Update* modela la operación de agregar información de un símbolo a la tabla, modificando los valores anteriores en caso de estar previamente cargados en la tabla. Por último, *LookUpOk* especifica la búsqueda de un elemento que se encuentra previamente cargado en la tabla (caso exitoso), mientras que *LookUpE* contempla el caso en el que el símbolo a buscar no se encuentre cargado (esquema de error de la operación).

2.2. Tácticas de testing y generación de clases de prueba.

La propuesta de Stocks y Carrington [SC96] es utilizar la especificación Z de una operación como fuente desde la cual obtener casos de prueba (abstractos) para testear el programa que supuestamente la implementa. La idea se basa en que la especificación del programa contiene todas las alternativas funcionales que el ingeniero consideró imprescindible describir para que el programador implemente el programa correcto. Por lo tanto, para saber si el programa funciona correctamente es necesario probarlo para cada una de esas alternativas funcionales. Entonces, más concretamente, la técnica se basa en expresar cada una de esas alternativas funcionales como un esquema de Z llamado *clase de prueba*.

El TTF comienza por definir, para cada operación Z, su espacio de entrada (*IS*, por *Input Space*) y a partir de este, su espacio válido de entrada (*VIS*, por *Valid Input Space*). El *IS* es el conjunto definido por todos los posibles valores de entrada y estado de la operación. Por ejemplo, el *IS* de *LookUp* es:

$$IS == [st : SYM \pfun VAL;\ s? : SYM]$$

El *VIS* es el subconjunto del *IS* para el cual la operación está definida, formalmente:

$$VIS_{Op} == [IS \mid pre\ Op]$$

En el caso de *LookUp* es igual a su *IS* ya que la operación es total.

Luego de determinar el *VIS*, el TTF propone dividir el mismo de modo tal que cada una de las particiones obtenidas represente una alternativa funcional distinta de la operación a testear. Llamaremos *clases de prueba* a estas particiones y serán el resultado de aplicar distintas tácticas de testing sobre el *VIS*. Podemos luego volver a aplicar tácticas sobre estas clases generadas y particionar nuevamente las mismas; este proceso se podrá repetir hasta que el ingeniero de testing considere que todas las alternativas funcionales importantes de la operación están representadas (cada una de estas alternativas corresponderá a una única clase de prueba). El último paso del proceso es la selección de al menos un *caso de prueba* para cada clase de prueba generada, esto cociste en buscar valores para las variables de la misma que reduzcan un predicado a verdadero. En general no haremos más referencia a este último paso ya que en este trabajaremos principalmente con la información contenida en las clases de prueba para obtener descripciones de los casos de prueba correspondientes.

Siguiendo con el ejemplo de *symbol table*, aplicaremos dos tácticas a *LookUp*. En primer lugar aplicaremos forma normal disyuntiva (DNF) que expresará la operación como una disyunción de esquemas en los cuales únicamente habrá conjunciones de negaciones de literales y luego dividirá el *VIS* con las precondiciones de cada esquema. Luego aplicaremos la táctica de Partición Estándar (SP) a la expresión: $s? \in \text{dom } st$ que dividirá el dominio del operador ($\in$ en este caso) según la partición propuesta por Stocks [**?**].

En la Figura 2.2 podemos ver el resultado de aplicar DNF y SP a la operación *LookUp*[1]. Cada una de las clases de prueba generadas representa una alternativa funcional en la cual deberemos testear el sistema, por ejemplo con un caso de prueba tomado de *LookUp_SP_1* estaremos probando el sistema en una situación en que el símbolo a buscar sea el único cargado en la tabla, así como con un caso de prueba de *LookUp_SP_4* estaremos testeando el sistema para el caso en el que el símbolo a buscar no se encuentra cargado en la tabla.

[1] Algunas clases de prueba generadas usando el TTF fueron obviadas ya que sus predicados contenían contradicciones. En estos casos no es posible obtener un caso de prueba a partir de ellas.

LookUp_DNF_1

LookUp_VIS

$s? \in \text{dom}\, st$

LookUp_SP_1

LookUp_DNF_1

$\text{dom}\, st = \{s?\}$

LookUp_SP_2

LookUp_DNF_1

$\text{dom}\, st \neq \{s?\}$

$s? \in \text{dom}\, st$

LookUp_DNF_2

LookUp_VIS

$s? \notin \text{dom}\, st$

LookUp_SP_4

LookUp_DNF_2

$\text{dom}\, st \neq \{s?\}$

$s? \in \text{dom}\, st$

Figura 2.2: Clases de prueba generadas para operación LookUp.

2.3. Fastest

Fastest[2] es una herramienta que implementa la teoría del TTF desarrollada en primer instancia por Maximiliano Cristiá y Pablo Rodriguez Monetti [CM09]. El desarrollo de la misma fue impulsado para intentar automatizar, lo mas posible, el proceso de testing funcional basado en especificaciones Z. *Fastest* se encuentra implementado mayormente en lenguaje Java haciendo uso de las librerías del framework CZT[3] (*Community Z Tools*) para contar con utilidades relacionadas al lenguaje de especificación Z.

Fastest fue utilizado en el pasado para testear el software a bordo de un satélite [CAF+11] y posteriormente se utilizaron técnicas de generación de lenguaje natural basada en templates para traducir los casos de prueba obtenidos [CP10]. Sin embargo, las técnicas utilizadas para esto eran dependientes del dominio de aplicación en cuestión.

Uno de los objetivos de este trabajo fue extender la implementación de *Fastest* para permitir al usuario generar traducciones o descripciones de las clases de prueba previamente generadas con la herramienta, y que las mismas resulten independientes del dominio de aplicación. El sistema de NLG desarrollado en este trabajo fue íntegramente implementado como una extensión de *Fastest*. Para esto hemos trabajado con la versión 1.6 de *Fastest*.

En el Apéndice X desarrollaremos algunos de los detalles mas importantes de esta implementación.

[2] http://www.flowgate.net/tools/Fastest.tar.gz
[3] http://czt.sourceforge.net/

Capítulo 3

Designaciones

Un modelo formal, como una especificación Z en este caso, es una abstracción de la realidad. Sin embargo, se realiza una especificación para escribir un programa que finalmente es usado en el mundo real. En consecuencia, existe una relación entre el modelo y la realidad. Normalmente en estos casos, cuando especificamos un sistema formalmente, es una practica común incluir asociaciones entre elementos de la especificación (operaciones, esquemas de estado, variables, constantes, etc.) y elementos que refieran al dominio de la aplicación. Estas asociaciones son llamadas *designaciones* [Jac95]. Sin esta documentación el modelo sería nada más que una teoría axiomática más sin conexión con la realidad.

Para documentar las designaciones usaremos una sintaxis similar[1] a la propuesta por Jackson [Jac95].

$$\textbf{termino\_formal} \approx \textit{texto en lenguaje natural}$$

El símbolo $\approx$ demarca la frontera entre el mundo formal o lógico (a la izquierda) y el mundo real (a la derecha). Del lado izquierdo estará el término formal a designar, este será un elemento de la especificación mientras que del otro lado tendremos texto informal en lenguaje natural que permitirá reconocer el fenómeno designado.

Continuando con el ejemplo de la tabla de símbolos (Figura 2.1), podríamos tener las siguientes designaciones para la operación *LookUp*:

LookUp	$\approx$ Se intenta buscar un símbolo en la tabla.
$s?$	$\approx$ El símbolo a buscar.
$st\ x$	$\approx$ Información asociada al símbolo x.

[1] A diferencia de Jackson escribiremos el término formal del lado izquierdo y el texto informal en lenguaje natural del otro lado.

Estas designaciones resultan de mucha utilidad. En primer instancia para cuando se empieza a escribir la especificación, la designación servirá para diferenciar un fenómeno en particular y darle un nombre. Luego, le será de utilidad al programador a la hora de leer la especificación. Por último, específicamente relacionado con los objetivos de este trabajo, estas designaciones nos resultarán un recurso primordial para la generación de descripciones independientes del dominio de aplicación.

Los sistemas de generación de lenguaje natural generalmente utilizan un repositorio de palabras o frases, las cuales se utilizan para referirse a fenómenos del dominio. En nuestro caso, el dominio de aplicación dependerá de la especificación en cuestión y de lo que se modele con la misma, por lo tanto, las designaciones resultarán nuestra única fuente de textos dependientes del dominio y es por eso que serán un elemento fundamental para nuestro trabajo.

3.1. Que designar en una especificación Z

Si bien Jackson [Jac95] propone designar la menor cantidad de fenómenos posibles y definir el resto en términos de estos, para el objetivo de éste trabajo puede ser beneficioso ser más flexibles en este aspecto. Por ejemplo, consideremos las siguientes designaciones para la especificación de un sistema para una base de datos de películas.

$Actor\ (x)$	$\approx x$ es un actor de cine.
$ActuoEn\ (x, y)$	$\approx$ El actor x actuaron en la película y.

Siguiendo lo propuesto por Jackson, si en la especificación apareciera el predicado:

ActuaronJuntosPelicula(x, y)

Que resulta verdadero para dos actores que hayan actuado juntos en alguna película. Éste podría definirse en términos las designaciones anteriores y, según Jackson, no debería designarse. Para nuestro caso en particular, para generar, por ejemplo, el texto:

"Samuel L. Jackson y Uma Thurman actuaron juntos en la película Pulp Fiction"

A partir de:

1. *ActuoEn("Samuel L. Jackson", "Pulp Fiction")*
2. *ActuoEn("Uma Thurman", "Pulp Fiction")*

Necesitaríamos procesar las designaciones teniendo en cuenta cuestiones referentes al dominio de aplicación. Y esto atentaría contra nuestros intereses de lograr un sistema de NLG independiente del dominio de aplicación.

Por otro lado creemos que también resultará de utilidad y otorgará al sistema una mayor flexibilidad permitir al usuario escribir designaciones que puedan resultar "redundantes" para la comprensión de la especificación pero de gran ayuda a la hora de generar una descripción en particular, como el ejemplo antes mencionado. En estos casos cuando nuestro sistema deba describir un término designado hará uso del texto incluido en la designación en lugar de intentar describir el mismo como veremos más adelante.

Finalmente, a partir de un análisis de textos generados para distintas especificaciones, creemos que, a fin de poder generar textos fluídos y naturales para el lector, es recomendable designar:

1. Nombres de esquemas de operación totales.
2. Variables de entrada/salida.
3. Variables de estado.
4. Tipos básicos.
5. Nombres de constructores de tipos libres.

Además, para los elementos modelados mediante una función es recomendable designar:

1. La función (f).
2. El dominio (dom f).
3. La aplicación (f x).

Capítulo 4

Generación de lenguaje natural.

La generación de lenguaje natural (NLG) es un una rama de la lingüística computacional y la inteligencia artificial encargada de estudiar la construcción de sistemas computacionales capaces de producir texto en español o cualquier otra lengua humana a partir de algún tipo de representación no-lingüística de la información a comunicar. Estos sistema combinan conocimientos tanto del lenguaje en cuestión cómo del dominio de aplicación para producir automáticamente documentos, reportes, mensajes o cualquier otro tipo de textos.

Dentro de la comunidad desarrolladora e investigadora de la NLG hay un cierto consenso sobre la funcionalidad lingüística general de un sistema de NLG. En este trabajo se optó por seguir la metodología más comúnmente aceptada, propuesta por Reiter y Dale [RD00]. A continuación describiremos brevemente los aspectos mas importantes de esta metodología y en capítulos posteriores desarrollaremos más en profundidad en los puntos mas relevantes para nuestro trabajo.

4.1. Análisis de requerimientos

El primer paso en la construcción de cualquier sistema de software, incluyendo los sistemas de generación de lenguaje natural, será el de realizar un análisis de requerimientos y a partir de ahí generar una especificación inicial del sistema.

Para el análisis de requerimientos, Reiter y Dale proponen realizar un *corpus* de textos de ejemplo y a partir de ellos obtener una especificación para el sistema a desarrollar. Estos ejemplos estarán compuestos por una colección de datos de entrada del sistema con sus respectivas salidas (texto en lenguaje natural). Estos deberán estar redactados por un humano experto y deberían caracterizar todas las salidas posibles que se espera que el sistema genere.

En el capítulo 5 profundizaremos más sobre este tema, describiremos y analizaremos el *corpus de descripciones* utilizado para este trabajo.

4.2. Tareas de la generación de lenguaje natural

Dentro de la comunidad desarrolladora e investigadora de la generación de lenguaje natural, hay cierto consenso sobre las tareas que deben llevarse a cabo para, a partir de los datos de entrada, generar texto final en lenguaje natural.

La más comúnmente aceptada es la clasificación de Reiter y Dale que distingue las siguientes siete tareas que deben ser realizadas a lo largo de todo el proceso:

Determinación del contenido: es el proceso de determinar que información debe ser comunicada en el texto final; será el encargado de que el mismo contenga toda la información requerida por el usuario. Generalmente involucra una o más tareas de selección, resumen y razonamiento con los datos de entrada.

Estructuración del documento: es el proceso de imponer un orden y estructura sobre los textos generados a fin de que la información del documento final se encuentre estructurada de forma entendible y fácil de leer.

Lexicalización: es el proceso de decidir que palabras y frases especificas usar para expresar los distintos conceptos y relaciones del dominio. En esta etapa se deberá establecer como se expresa un significado conceptual concreto, descrito en términos del modelo del dominio, usando elementos léxicos (sustantivos, verbos, adjetivos, etc).

Generación de expresiones de referencia: es la tarea de elegir que expresiones usar para identificar entidades del dominio de aplicación. Podríamos querer referirnos a una determinada entidad de distintas formas. Por ejemplo: podríamos querer referirnos al mes en curso como, “febrero”, “este mes”, “este”, etc.

Agregación: se encarga de combinar dos o mas elementos informativos con el fin de conseguir un texto más fluido y legible. La agregación decide que elementos se pueden agrupar para generar oraciones mas complejas sin modificar el significado de las mismas. Por Ejemplo, dos frases de una descripción para una clase de prueba de un scheduler se podrían expresar como: *“El proceso a borrar se encuentra en la tabla de procesos. El estado del proceso a borrar es waiting.”* o *“El proceso a borrar se encuentra en la tabla de procesos y el estado del mismo es waiting.”*

Realización lingüística: es el proceso de aplicar reglas gramaticales (a estructuras generadas por las etapas anteriores) con el fin de producir un texto que sea sintáctica, morfológica y ortográficamente correcto.

Realización de la estructura: esta tarea se encarga de convertir estructuras abstractas como párrafos y secciones (generadas por etapas anteriores) en texto comprensible por el componente de presentación del documento. Por ejemplo, la salida del sistema de NLG podría ser código LaTeX para luego ser post-procesado, en este caso sería esta etapa la encargada de agregar delimitadores y comandos de LaTeX para generar el documento.

4.3. Arquitectura para la NLG.

Existen muchas maneras de construir un sistema que realice las tareas antes mencionadas. Una forma podría ser construir un único módulo encargado de llevar a cabo todas las tareas en simultaneo. En el otro extremo, podríamos tener un módulo separado para cada tarea y conectarlos mediante un *pipeline*. En este caso, el sistema primero se encargaría de la determinación de contenido, luego la estructuración del documento, y así sucesivamente. La desventaja de este último modelo es que asume que las tareas deben ser realizadas en un único orden y que las funcionalidades de las mismas no se solapan.

En este trabajo utilizaremos la arquitectura mas comúnmente utilizada para sistemas de NLG. Esta consiste de tres módulos conectados mediante un *pipeline*. La misma se encuentra en el medio de los dos extremos antes mencionados.

El primer módulo de nuestro sistema será el *document planner*, encargado de realizar las tareas de determinación de contenido y estructuración del documento. Veremos en el capítulo 6 que estas tareas se encuentran sumamente relacionadas y son realizadas en simultaneo. La salida de esta etapa y entrada del *microplanner* será un *document plan*, este será una abstracción del documento final que contendrá los elementos informativos que se desea comunicar.

El segundo módulo será el encargado de realizar las tareas de lexicalización, generación de expresiones de referencia y agregación. La función del mismo será la de trabajar el document plan generando una especificación mas refinada del texto final, una *text specification*. El *microplanner* será el responsable de transformar los elementos informativos incluidos en el document plan en una especificación mas concreta de una oración. En el capítulo 7 desarrollaremos mas en profundidad el funcionamiento del microplanner.

Finalmente el *surface realiser* tomará como entrada el text specification generado por la etapa anterior y será el encargado de producir el texto final. Este módulo deberá llevar a cabo las tareas de realización lingüística y de superficie ates mencionadas.

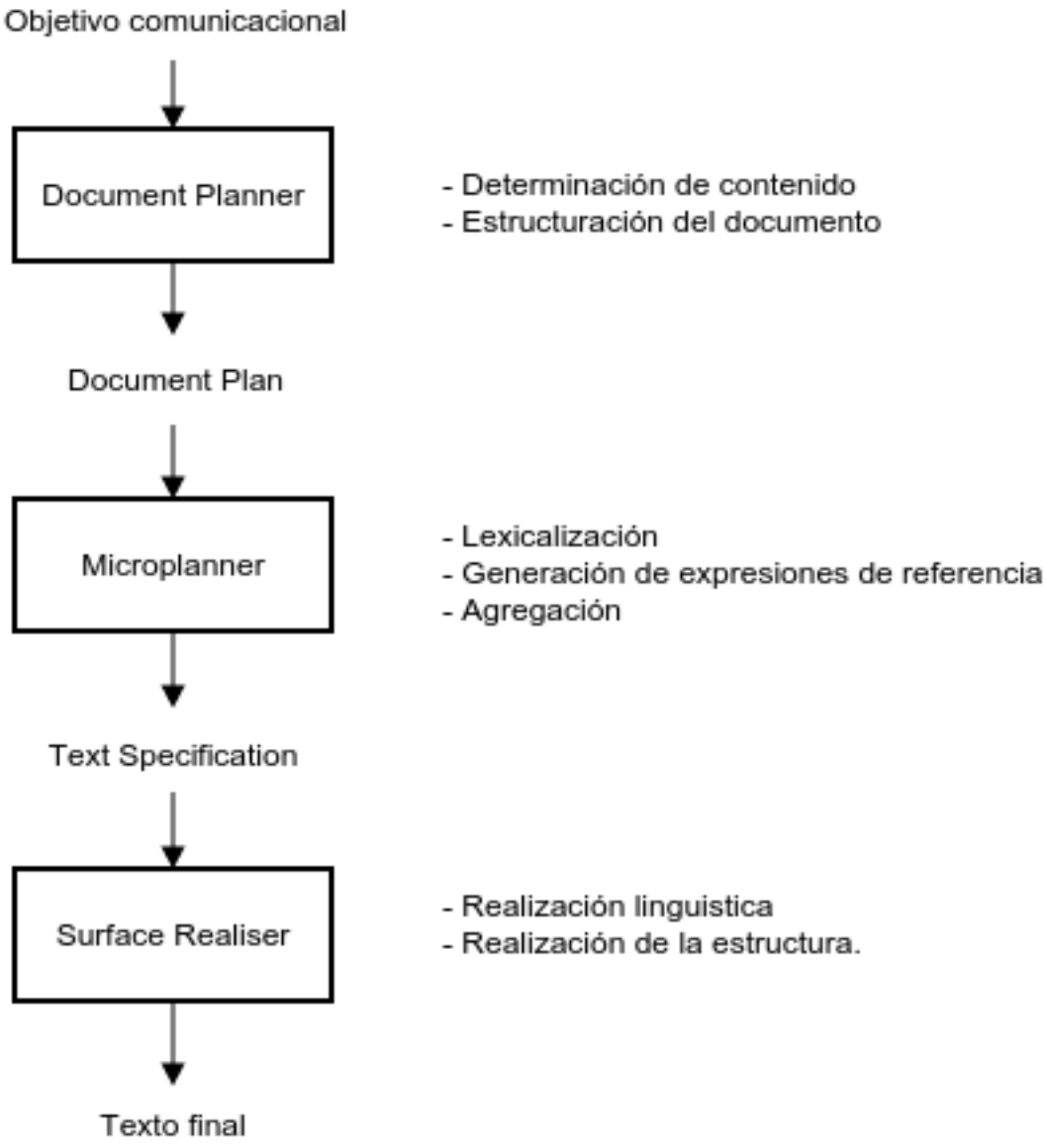

Figura 4.1: Arquitectura típica sistema NLG.

Capítulo 5

Análisis de requerimientos

El primer paso en la construcción de cualquier sistema de software, incluyendo los sistemas de generación de lenguaje natural, será el de realizar un análisis de requerimientos y a partir de ahí generar una especificación inicial del sistema.

Para el análisis de requerimientos, seguiremos el enfoque propuesto por Reiter y Dale [RD00] en el cual se propone realizar un *corpus* de textos de ejemplo y a partir de ellos obtener una especificación para nuestro sistema.

5.1. Corpus de descripciones

Este *corpus de textos* constará de una colección de ejemplos, formados por la entrada y la salida esperada de nuestro sistema. En nuestro caso, la entrada de nuestro sistema será: la especificación formal en lenguaje Z, un conjunto de designaciones y un grupo de clases de prueba generadas de antemano, mientras que la salida estará formada por descripciones en lenguaje natural de las clases de prueba antes mencionadas. En lo posible, el *corpus* de textos debe cubrir todo el rango de textos esperados a ser producidos por el sistema de NLG; éste debería cubrir los casos más frecuentes, así como los casos mas inusuales que se puedan dar.

Siguiendo la metodología propuesta por Reiter y Dale, para obtener un *corpus objetivo* deberíamos recolectar un conjunto lo suficientemente amplio de ejemplos a fin de caracterizar la variedad de textos que deseamos generar. Luego una persona capacitada (en nuestro caso sería una persona capaz de leer Z) debería describir en lenguaje natural los ejemplos antes mencionados. Finalmente se debería revisar este *corpus inicial* y modificarlo en caso de existir algún algún texto que resulte técnicamente imposible de generar o prohibitivamente caro a nivel computacional. Una vez finalizado este proceso tendríamos en nuestro poder un *corpus objetivo* que nos servirá para sustentar muchas de las decisiones que tomaremos a lo largo de este trabajo. Ésta colección de ejemplos también nos servirá para realizar una evaluación de nuestra

implementación, comparando los textos generados por nuestro sistema con las descripciones del *corpus* realizadas manualmente por una persona.

En el apéndice A se pueden consultar los textos incluidos en el *corpus* utilizado para este trabajo. Para elaborar el mismo recolectamos una serie de clases de prueba generados con *Fastest* a partir de distintas especificaciones y luego escribimos manualmente cada una de las descripciones de estas clases de prueba. Con las especificaciones y clases de prueba incluidas en el *corpus*, intentamos abarcar todo el rango de textos que esperamos que nuestro sistema sea capaz de producir, para esto tuvimos en cuenta incluir clases de pruebas que cubran todas las expresiones de Z contempladas dentro del alcance de este trabajo y sus posibles combinaciones. También trabajamos con especificaciones sobre distintos dominios de aplicación con el fin de lograr *corpus* que nos sea de utilidad para dar con una solución independiente del dominio de aplicación. En total trabajamos con clases de prueba generadas con *Fastest* para 10 especificaciones distintas; del total de clases de prueba correspondientes a estas especificaciones se escogieron las más significativas para describir y se ignoraron aquellas que contenían algún operador no considerado dentro del alcance de este trabajo.

La Figura 5.2 muestra a modo de ejemplo una descripción para la clase de prueba *LookUp_SP_1* generada a partir de la especificación para una tabla de símbolos introducida anteriormente (pág. 12). El *corpus de descripciones* utilizado para este trabajo, constará entonces de una colección de ejemplos similares a éste.

LookUp_SP_1

LookUp_VIS

$s? \in \operatorname{dom} st$

$\operatorname{dom} st = \{s?\}$

Figura 5.1: Clase de prueba para operación LookUp.

Descripciones SymbolTable

LookUp_SP_1: Se busca un símbolo en la tabla.

- Cuando:
  - El símbolo a buscar pertenece a los símbolos cargados en la tabla de símbolos.
  - El símbolo a buscar es el único elemento del conjunto formado por los símbolos cargados en la tabla de símbolos.

Figura 5.2: Descripción en lenguaje natural para *LookUp_SP_1*.

5.2. Análisis del corpus

Como mencionamos previamente muchas decisiones de diseño de nuestro sistema estarán fundamentados en observaciones obtenidas a partir del *corpus*, mediante un análisis del mismo podremos obtener una especificación para nuestro sistema d NLG.

Lo primero que podemos observar de los textos de ejemplo es la estructura del texto que se mantiene en la descripción de cada clase de prueba. En todas, se comenzará por el nombre de la clase de prueba, seguido por un pequeño detalle de la operación a testear y luego una lista de oraciones, una por cada restricción de la clase de prueba que se está describiendo. Esto nos ayudará en el capítulo 6 para definir la estructura de nuestro documento.

El desafío principal de nuestro sistema será describir en lenguaje natural las distintas expresiones Z que pueden aparecer en el cuerpo de los esquemas de una clase de prueba. Nuestro sistema de NLG deberá ser capaz de construir una oración en lenguaje natural a partir de una expresión Z y un conjunto de designaciones. En lo que queda de esta sección nos concentraremos en la tarea de *verbalizar* expresiones Z; definir con precisión los detalles de la misma será de vital importancia para especificar las etapas de *microplanning* y *lexicalización* de nuestro sistema en los capítulos 7 y 8.

Siguiendo con el análisis del *corpus*, podemos notar, como era de esperarse, que los textos que surgen a partir de los operadores de Z se repiten (con pequeñas variantes que analizaremos mas adelante) independientemente del dominio de aplicación; y las diferencias que aparecen entre los mismos se deben principalmente al texto incluido en las designaciones. Por ejemplo, consideremos las siguientes dos expresiones y sus respectivas descripciones en lenguaje natural, una pertenece al ejemplo anterior y la otra la otra forma parte de una clase de prueba para la especificación de un sistema bancario:

1. $s? \in \text{dom} \rightarrow$ *"El símbolo a buscar* ***pertenece a*** *los símbolos cargados en la tabla de símbolos."*
2. $s? \in \text{dom} \rightarrow$ *"El número de cuenta* ***pertenece a*** *los números de cuenta cargados en el banco."*

Como vemos, el texto *"pertenece a"* aparece en ambas descripciones como resultado de verbalizar el operador $\in$ de Z y como mencionamos antes, se diferencian en el texto que antecede y precede al anterior en base a descripciones generadas a partir de las designaciones de cada sistema. Podríamos entonces intentar definir la tarea de verbalización mediante una función que tome como entrada una expresión Z y devuelva una descripción en lenguaje natural para la misma. La definición de esta función dependerá del argumento, y para el caso anterior podría ser la siguiente:

$$verbalizar(x \in y) \rightarrow verbalizar(x) + \text{"pertenece a"} + verbalizar(y)$$

A partir de esto podemos ver que para verbalizar términos compuestos de Z, necesitaremos verbalizar recursivamente las partes que los componen, en la definición anterior necesitaríamos conocer las verbalizaciones de las expresiones x e y para poder obtener una verbalización para $x \in y$.

Retomemos el ejemplo de la figura 5.1, nos concentraremos en verbalizar la primera expresión de la clase de prueba:

$$s? \in \operatorname{dom} st$$

En este caso, según la verbalización propuesta anteriormente, podríamos verbalizar la expresión como:

$$\mathit{verb}(s? \in \operatorname{dom} st) \rightarrow \mathit{verb}(s?) + \text{“}\mathit{pertenece\ a}\text{”} + \mathit{verb}(\operatorname{dom} st)$$

Teniendo que verbalizar las expresiones $s?$ y $\operatorname{dom} st$. En este caso, ambas expresiones se encuentran designadas. En estas situaciones nuestra tarea de verbalización debería construir la descripción en base al texto presente en la designación y no intentar describir la expresión en base al término en cuestión. En el ejemplo anterior, no deberíamos intentar verbalizar el término $\operatorname{dom} st$ como *“el dominio”* + $\mathit{verb}(st)$, por ejemplo, ya que estaríamos perdiendo información valiosa para nuestras descripciones contenida en las designaciones. Entonces, será un requerimiento para nuestra tarea de verbalización contemplar en primera instancia si la expresión a describir no se encuentra designada antes de intentar describir el término Z correspondiente; de estar designada, se deberá construir una descripción en base a su designación.

En la figura 5.3 podemos ver un algoritmo en pseudocódigo para nuestra tarea de verbalización, siendo la función *verb* la encargada de generar las verbalizaciones para las expresiones de Z en base a un conjunto de reglas dependientes de el término a describir, como la introducida anteriormente para el caso del operador $\in$.

```
function VERBALIZACION(exp)
    if esta_designada(exp) then
        ret ← designacion(exp)
    else
        ret ← verb(exp)
    end if
    return ret
end function
```

Figura 5.3: Bosquejo verbalización.

En el bosquejo anterior, abstraemos mediante la función *designacion* la tarea de generar una descripción para una expresión designada en base al

texto presente en la misma. Cabe aclarar, que en este caso, tendremos que contemplar si la designación correspondiente es una designación parametrizada. De no ser el caso, el resultado sería exactamente el texto que forma parte de la designación, del contrario, habrá primero que verbalizar el parámetro de la designación, teniendo luego que construir la descripción final a partir del texto de la designación parametrizada y la verbalización del parámetro.

Finalmente, presentaremos un conjunto de reglas para la verbalización de expresiones, construido a partir de un análisis de los textos de ejemplo presentes en el *corpus*.

▷ $=$

1. $\{exp1\} = exp2$
   → `verb(exp1)` + "*es el único elemento de*" + `verb(exp2)`
2. $exp1 = \{\}$
   → "*no hay ningún elemento en*" + `verb(exp1)`
3. $exp1 \cap exp2 = \{\}$
   → `verb(exp1)` + "*y*" + `verb(exp2)` + "*no tienen ningún elemento en común*"
4. $exp1 \cap \{exp2\} = \{\}$
   → `verb(exp2)` + "*no pertenece a*" + `verb(exp1)`
5. $exp1 = exp2$ (caso default)
   → `verb(exp1)` + "*es(son) igual(iguales) a*" + `verb(exp2)`

▷ $\neq$

1. $exp1 \neq \{\}$
   → "*existe al menos un elemento en*" + `verb(exp1)`
2. $exp1 \cap exp2 \neq \{\}$
   → `verb(exp1)` + "*y*" + `verb(exp2)` + "*tienen al menos un elemento en común*"
3. $exp1 \neq exp2$ (caso default)
   → `verb(exp1)` + "*no es(son) igual(iguales) a*" + `verb(exp2)`

▷ $<$

1. $exp1 < exp2$ (caso default)
   → `verb(exp1)` + "*es menor a*" + `verb(exp2)`
2. $exp1 < 0$
   → `verb(exp1)` + "*es negativo*"

▷ $\leq$

1. $exp1 \leq exp2$ (caso default)
   → `verb(exp1)` + "*es menor o igual a*" + `verb(exp2)`

▷ $>$

1. $exp1 > exp2$ (caso default)

→ `verb(exp1)` + "*es mayor a*" + `verb(exp2)`

2. $exp1 > 0$
   → `verb(exp1)` + "*es positivo*"

▷ $\geq$

1. $exp1 \geq exp2$ (caso default)
   → `verb(exp1)` + "*es mayor o igual a*" + `verb(exp2)`

▷ $\in$

1. $exp1 \in exp2$ (caso default)
   → `verb(exp1)` + "*pertenece(n) a*" + `verb(exp2)`

▷ $\notin$

1. $exp1 \notin exp2$ (caso default)
   → `verb(exp1)` + "*no pertenece(n) a*" + `verb(exp2)`

▷ $\subset$

1. $exp1 \subset exp2$ (caso default)
   → `verb(exp1)` + "*está(n) incluido/a(s) en*" + `verb(exp2)`

2. $\{exp1, exp2, \ldots, expn\} \subset expm$ (caso default)
   → `verb(exp1)` + "," + `verb(exp2)` + "*, . . . , y*" + `verb(expn)` + "*pertenece(n) a*" + `verb(expm)`

▷ $\not\subset$

1. $exp1 \not\subset exp2$ (caso default)
   → `verb(exp1)` + "*no está(n) incluido/a(s) en*" + `verb(exp2)`

▷ $\subseteq$

1. $exp1 \subseteq exp2$ (caso default)
   → `verb(exp1)` + "*está incluido o es igual a*" + `verb(exp2)`

2. $\{exp1, exp2, \ldots, expn\} \subseteq exp2$
   → ídem inclusión

▷ $\not\subseteq$

1. $exp1 \not\subseteq exp2$ (caso default)
   → "*existe al menos un elemento en*" + `verb(exp1)` + "*que no se está en*" + `verb(exp2)`

▷ $\mapsto$

1. $exp1 \mapsto exp2$ (caso default)
   → "*el par ordenado formado por:*" + `verb(exp1)` + "*y*" + `verb(exp2)`

▷ $\{a, b, \ldots\}$

1. $\{\}$
   → "*el conjunto vacío*"

2. $\{exp1\}$
   → "*el conjunto formado por* " + `verb(exp1)`

3. $\{exp1, exp2, \ldots, expn\}$ (caso default)
   → "*el conjunto formado por*" + `verb(exp1)` + "," + `verb(exp2)` + "*, . . . , y*" + `verb(expn)`

▷ $\cup$
1. $exp1 \cup exp2$ (caso default)
→ "*elementos en*" + `verb(exp1)` + "*y en*" + `verb(exp2)`

▷ $\cap$
1. $exp1 \cap exp2$ (caso default)
→ "*elementos en*" + `verb(exp1)` + "*que también se encuentren en*" + `verb(exp2)`

▷ $f\ x$ (aplicación)
1. $f\ exp1$ (caso default)
→ `verb(f)` + "*aplicada a*" + `verb(exp1)`

▷ *dom*
1. $dom(exp1)$ (caso default)
→ "*dominio de*" + `verb(exp1)`

▷ *ran*
1. $ran(exp1)$ (caso default)
→ "*rango de*" + `verb(exp1)`

Como era de esperarse, podemos observar que, a fin de lograr descripciones mas naturales, deberemos contemplar algunas combinaciones entre los distintos términos posibles. Por ejemplo, podemos ver que en el caso de la igualdad entre conjuntos, se propone una descripción para el caso en que uno de los conjuntos esté formado por un único elemento y otra verbalización distinta en el caso de que éste conjunto se se encuentre vacío.

Otro punto a tener en cuenta al momento de verbalizar una expresión será que las designaciones documentadas por el usuario pueden no contener un artículo que acompañe al sustantivo. En estos casos nuestro sistema deberá identificar los casos en los que sea necesario y agregar el artículo apropiado de forma que concuerde en género y número con el sustantivo.

Finalmente, en base a las reglas introducidas previamente, podemos notar que la mayoría de las frases a generar se tratan de oraciones bimembres ordenadas como: *sujeto + verbo + objeto*. Todas expresadas en tiempo presente. Además, en algunos casos será necesario conjugar el verbo de una oración de forma tal que concuerde con el número del sujeto de la misma. Algo parecido pasa con el atributo en los casos que utilizamos un verbo copulativo en el que deberemos hacer concordar el mismo con número y género del sujeto. Estas últimas observaciones deberemos tenerlas en cuenta para el desarrollo de nuestro realizador lingüístico.

Capítulo 6

Document Planning

En la arquitectura presentada en el capítulo 4 mencionamos que el *document planner* es el responsable de decidir que información comunicar (*determinación de contenido*) y como deberá estar estructurada esta información en el texto final (*estructuración de documento*). El document planner será el encargado de que el documento final contenga toda la información requerida por el usuario y que la misma se encuentre estructurada de una forma razonablemente coherente. El resultado de esta etapa será un *document plan* en el cual se especifica qué contenido debe ser incluido en el texto final y de que forma debe estar estructurado.

A continuación describiremos brevemente la entrada y salida de nuestro *document planner*, definiremos como modelar los elementos informativos (estos serán elementos de nuestro *document plan*) y finalmente describiremos las tareas de *determinación de contenido* y *estructuración de documento*.

6.1. Entrada y salida del document planner

Como el document planner es el primer módulo del pipeline, la entrada del document planner será la misma que la entrada de nuestro sistema. Reiter y Dale [RD00] generalizan la entrada de un sistema de NLG como una cuádrupla compuestas por los siguientes componentes:

Fuente de conocimiento: Se refiere a las bases de datos e información del dominio de aplicación que nos proporcionará el contenido de la información que los textos generados deberán contener. En nuestro caso la fuente de conocimiento estará compuesta por la especificación, las designaciones de la misma y las clases y casos de prueba generados.

Objetivo comunicacional: Especifica el propósito que debe cumplir el sistema. En general esta compuesto por un "tipo de objetivo" y un parámetro. En este trabajo tendremos solo un tipo de objetivo comunicacional: *Describir(x)*,

dónde el parámetro x será un conjunto de identificadores de las clases de prueba que desean describirse.

Modelo de usuario: Provee información acerca del usuario (nivel de experiencia, preferencias, etc.). En nuestro caso el sistema se comportará de la misma forma independientemente del usuario, por lo que no tendremos en cuenta información del mismo.

Historial de discurso: Consta de información sobre interacciones previas entre el usuario y el sistema. Este historial puede servir para algunos sistemas interactivos donde las interacciones previas con el usuario pueden resultar de utilidad.

La salida del document planner será un document plan. En nuestra arquitectura el document plan está estructurado como un árbol, donde las hojas representan el contenido y los nodos internos especifican información estructural, por ejemplo sobre como debe agruparse la el contenido a comunicar. En la sección 6.4 desarrollaremos este tema mas en detalle.

6.2. Representación del dominio

En los sistemas de NLG el texto generado se utiliza principalmente para transmitir información. Esta información será expresada generalmente en frases y palabras, pero estas frases y palabras no son en si mismo la información; la información subyace estos constructores lingüísticos y es "llevada" por ellos. Nos deberemos concentrar entonces en como representar este conocimiento y como "mapear" estas estructuras a una representación semántica.

El *corpus de descripciones* (apéndice **??**) resulta una buena fuente para estudiar y definir como deberemos representar la información o *mensajes*[1] a comunicar. Podemos ver en todos los ejemplos del *corpus* una relación directa entre la información a comunicar y las expresiones Z pertenecientes al dominio de aplicación, donde podemos observar que cada linea de una descripción se corresponde perfectamente con la verbalización de la expresión en lenguaje Z correspondiente. Veamos por ejemplo las siguientes líneas de la descripción de LookUp_SP_1 introducida anteriormente (pág. 26):

1. *"El símbolo a buscar pertenece a los símbolos cargados en la tabla de símbolos."*
2. *"El símbolo a buscar es el único elemento del conjunto formado por los símbolos cargados en la tabla de símbolos."*

[1] Reiter y Dale llaman *mensajes* a elementos informativos que conceptualizan la información que queremos comunicar. Estos están compuesto en sí mismos por elementos del dominio de aplicación.

en las que podemos observar que la información de cada linea de la descripción se encuentra perfectamente caracterizada por la expresión Z que describe, enumeradas a continuación:

1. $s? \in \text{dom } st$
2. $\text{dom } st = \{s?\}$

Debido a esta correlación entre las expresiones de las clases de prueba y la información a generar, la verbalización de las mismas, podemos establecer un único tipo de *mensaje* para nuestro sistema: *VerbalizacionExpresion*. Este representa, como su nombre lo indica, una verbalización de una expresión Z. En la figura 6.1 podemos ver como quedarían definidos los mensajes para las dos lineas de la descripción de *LookUp_SP_1* vista anteriormente.

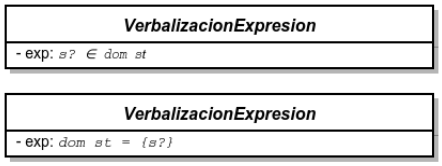

Figura 6.1: Mensajes a comunicar para el ejemplo de la figura 5.2.

6.3. Determinación del contenido

La determinación del contenido es el nombre que se le da a la tarea de decidir y obtener la información que se debe comunicar en un texto. Este proceso generalmente involucra una o más tareas de selección, resumen y razonamiento con los datos de entrada. El *proceso de selección* recopilará un subconjunto de la información de entrada para luego poder ser comunicada al usuario. El objetivo del mismo será el de proveer la información relevante requerida por el mismo. La tarea de *resumen* es necesaria cuando los datos de entrada son muy "granulados" para ser comunicados directamente o si la información relevante consiste alguna generalización o abstracción de los mismos, que no es el caso de nuestro sistema donde las expresiones de Z con las que trabajaremos contienen exactamente la información que se desea comunicar. Por último el *razonamiento con los datos* resulta un caso general de las dos anteriores. Finalmente, una vez seleccionada y procesada la información necesaria será esta etapa la encargada de construir los mensajes introducidos en la sección anterior, que luego formarán parte de nuestro *document plan*.

Para nuestro trabajo, la tarea de selección se resume en la búsqueda y filtrado de las clases de prueba indicadas por el usuario dentro de todo el conjunto de clases de prueba que forma parte de la entrada de nuestro sistema. Por ejemplo, si deseamos generar una descripción para la clase de prueba *LookUp_SP_1* de la figura 2.2, la misión de de esta tarea será la de identificar

y seleccionar la clase de prueba *LookUp_SP_1* entre todas las clases de pruebas que forman parte de los datos de entrada de nuestro sistema de NLG.

Luego de la selección, nuestro sistema deberá procesar los datos de entrada con el fin de obtener mejores descripciones. Hemos observado[2] que trabajando ciertas expresiones en las clases de prueba podemos mejorar considerablemente los textos generados. Veamos por ejemplo la figura 6.2 donde se muestra una clase de prueba generada con *Fastest* en la que se pueden ver dos problemas que abordaremos en esta etapa.

Update_SP_4

$$st : SYM \pfun VAL$$
$$s? : SYM$$
$$v? : VAL$$

$$st \neq \{\}$$
$$\{s? \mapsto v?\} \neq \{\}$$
$$\mathrm{dom}\, st = \mathrm{dom}\{s? \mapsto v?\}$$

Figura 6.2: Clase de prueba para operación Update (pág. 12).

Podemos observar que la siguiente expresión del ejemplo anterior:

$$\{s? \mapsto v?\} \neq \{\}$$

no aporta información importante para el usuario, de hecho esta expresión no agrega ninguna restricción para el caso de prueba ya que será siempre verdadera. De no filtrar esta expresión tempranamente, terminaríamos como resultado un texto generando parecido al siguiente:

"el conjunto formado por el par de el símbolo a actualizar y el nuevo valor, es distinto al conjunto vacío"

que además de resultar algo difícil de interpretar, no aporta nada al objetivo comunicacional.

Por otro lado, en un primer intento por describir automáticamente la expresión:

$$\mathrm{dom}\, st = \mathrm{dom}\{s? \mapsto v?\}$$

podríamos describirla como[3]:

[2] Las observaciones son en base a clases de prueba generados utilizando *Fastest 1.6*

[3] Esta descripción sería la generada utilizando el sistema de reglas propuesto en el capítulo **??** si no trabajamos la expresión en una etapa previa.

"el conjunto de símbolos cargados en la tabla es igual a el dominio del par formado por el símbolo a actualiza y el nuevo valor"

Es posible simplificar notablemente esta descripción si antes trabajamos la expresión anterior, que resulta equivalente a:

$$\mathrm{dom}\, st = \{s?\}$$

que luego podríamos describir como:

"el símbolo a actualizar es el único elemento en la tabla de símbolos cargados"

En conclusión, el procesamiento que nos proponemos a realizar en esta etapa tendrá dos objetivos:

1. Eliminar tautologías de las expresiones que forman parte de las clases de prueba seleccionadas.
2. Realizar algunas simplificaciones o reducciones triviales.

Una vez seleccionada y procesada la información deberemos construir los *mensajes* (*VerbalizacionExpresion*) que luego formarán parte del *document plan* desarrollado en el siguiente capítulo.

6.4. Estructuración del documento

Como dijimos antes, el texto generado no podrá ser una colección al azar de frases y palabras. Deberá tener coherencia y poseer una estructura que le permita al lector interpretar con facilidad el contenido del mismo. Necesitaremos considerar como organizar y estructurar la información que debemos comunicar con el fin de producir un texto razonablemente fácil de leer y comprender.

En esta tarea nos concentraremos en construir una estructura que contenga los *mensajes* seleccionados en la etapa de *determinación de contenido*; estableciendo el agrupamiento y ordenamiento de los mismos. Esta estructura deberá caracterizar la disposición de los elementos pertenecientes a los textos recopilados en el *corpus*.

Tomando el *corpus* de descripciones como una especificación de los documentos que debemos generar podemos observar que estos documentos poseen una estructura bastante simple y rígida a la vez. Estos documentos deben estar formados por una secuencia de descripciones para las clases de prueba indicadas por el usuario, ordenadas alfabéticamente según el nombre de la clase de prueba. A su vez, cada una de estas descripciones deberá agrupar las verbalizaciones de las expresiones seleccionadas en la etapa de *determinación de contenido*, ordenadas de la misma forma en la que aparecen en el

esquema de la clase de prueba en cuestión. En la figura 6.3 podemos observar una representación abstracta de la estructura propuesta para modelar el documento.

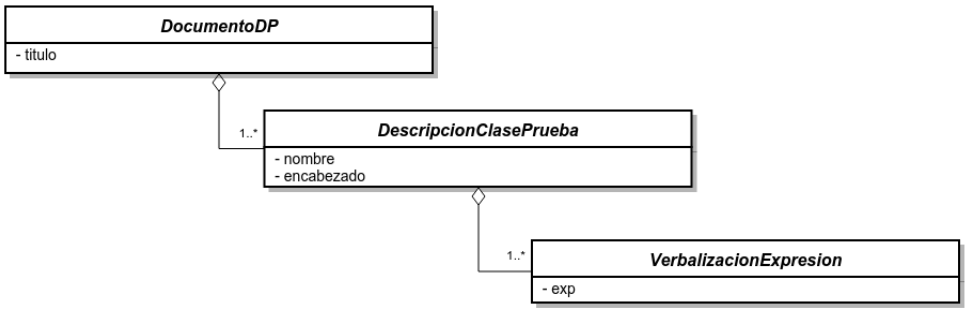

Figura 6.3: Document plan.

Llamaremos *DocumentoDP* a la raíz de nuestro *document plan*, *DocumentoDP* contendrá a su vez una lista ordenada de las descripciones de las clases de prueba (*DescripcionClasePrueba*) que debemos incluir en el texto final. El elemento *DescripcionClasePrueba* representa el texto a generar para una clase de prueba (por ejemplo el texto de la figura 5.1) y tendremos uno de estos elementos por cada clase de prueba indicada por el usuario. Finalmente los mensajes seleccionados en la etapa anterior formarán se encontrarán agrupados en la *DescripcionClasePrueba* correspondiente. Podemos ver en la figura 6.4 un ejemplo del document plan para la descripción de la clase de prueba *LookUp_SP_1* introducida anteriormente.

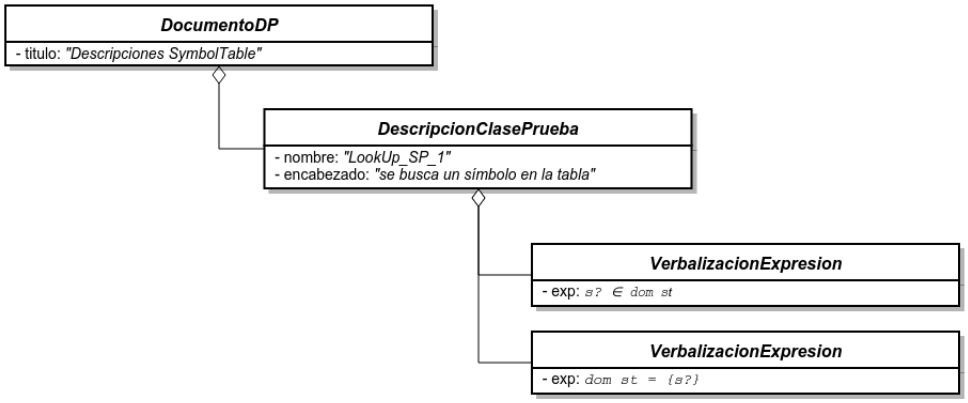

Figura 6.4: Document plan correspondiente al texto de la figura 5.1.

Capítulo 7

Microplanning

En éste capítulo describiremos las tres tareas involucradas en el proceso de microplanning: lexicalización, agregación y generación de expresiones de referencia. Luego definiremos en detalle la entrada y salida de esta etapa. Finalmente profundizaremos sobre la tarea de *lexicalización*, que será la única de las tareas antes mencionadas realizada por nuestro mircroplanner.

7.1. Tareas del Microplanner

La tarea del microplanner será tomar el document plan generado en la etapa anterior y refinarlo a modo de producir una especificación mas detallada del texto a generar. Cabe aclarar que el resultado de esta etapa no será todavía el texto final, sino que quedarán por tomar decisiones acerca de la sintaxis, morfología y cuestiones de presentación, de las cuales se encargará el *realizador de superficie*.

Como mencionamos en el capítulo 4, las tareas que debería realizar el microplanner son:

Lexicalización. Esta tarea se encargará de elegir que palabras particulares, constructores sintácticos usar para comunicar la información contenida en el document plan. Desarrollaremos más en detalle el trabajo realizado por esta etapa en la sección 7.3

Agregación. Esta tarea deberá combinar elementos informativos con el fin de conseguir un texto más fluido y legible. La agregación decide que elementos se pueden agrupar para generar oraciones mas complejas sin modificar el significado de las mismas. Por Ejemplo, dos frases de una descripción para una clase de prueba de un scheduler se podrían expresar como:

1. *"El proceso a borrar se encuentra en la tabla de procesos. El estado del proceso a borrar es waiting."*
2. *"El proceso a borrar se encuentra en la tabla de procesos y el estado del mismo es waiting."*

Decidimos para este trabajo expresar nuestras descripciones siguiendo el estilo de la primer frase del ejemplo anterior, es por esto que nuestro microplanner no realizará tareas de agregación. En nuestro caso en particular creemos que es útil para el lector que cada frase de nuestra descripción haga referencia a una única restricción del esquema de la clase de prueba. De esta forma podríamos identificar con mayor facilidad cual es la descripción para una expresión particular de la clase de prueba.

Generación de expresiones de referencia. Esta tarea se encarga de determinar que frases deben ser usadas para identificar las diferentes menciones al mismo elemento en un texto a fin de aportar fluidez al mismo. Por ejemplo, en los casos que se hace referencia a una entidad que ya ha aparecido en el texto (referencia posterior) se puede remplazar la misma por otra frase que la referencie (generalmente un sintagma nominal). La elección de qué expresión usar para referirse a la entidad dependerá del contexto y deberá hacerse sin generar ambigüedad para el lector. Por ejemplo, siguiendo con el ejemplo anterior del scheduler, podríamos reemplazar la segunda ocurrencia de "el proceso a borrar" en la primer frase por el pronombre "mismo", quedando entonces:

"El proceso a borrar se encuentra en la tabla de procesos. El estado del mismo es waiting."

Nuestro microplanner no realizará tareas de generación de expresiones de referencia ya que se encuentran fuera del alcance de este trabajo. Además, como podemos observar en el *corpus* nuestras descripciones de clases de prueba están formadas por una serie de oraciones individuales, donde cada una de estas describe una restricción de la clase de prueba dada; estas oraciones resultan relativamente concisas y es extraño que hagan referencia en más de una oportunidad a un mismo elemento, por lo tanto creemos que no resulta indispensable contar con un generador de expresiones de referencia en nuestro trabajo.

7.2. Entrada y salida del microplanner

Como vimos en el capítulo anterior, la salida del document planner será una estructura donde se encuentran agrupados los elementos informativos que deseamos comunicar. Estos elementos o *mensajes* especifican de una manera abstracta qué debemos comunicar en el texto final, pero no especifican, por ejemplo, que palabras debemos usar para hacerlo. Será el microplanner quien deberá tomar el document plan y producir una especificación mas refinada del texto que deseamos generar.

Esta especificación del texto tendrá también una estructura de árbol, donde los hojas especificarán las frases u oraciones a generar (*phrase specification*) y los nodos internos establecerán cómo estas frases tendrán que ser agrupadas

en elementos del documento (*text specification*) como párrafos, secciones, lista de items, etc. Luego, será tarea de la etapa de realización convertir los nodos internos en anotaciones especificas para el sistema de presentación (realización de estructura) y transformar las *phrase specification* en oraciones o frases sintáctica, morfológica y ortográficamente correctas (realización lingüística).

En nuestro caso contaremos con sólo dos elementos para modelar las estructura interna del documento: *TSDocumento* y *TSListaItems*. El primero especificará la raíz del documento y contendrá un conjunto de *TSListaItems* que modelan la lista de descripciones a generar para cada clase de prueba.

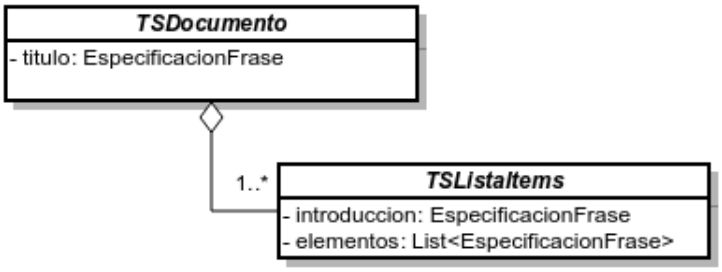

Figura 7.1: Text Specification.

Luego en la etapa de *realización de estructura* se deberán transformar estas estructuras en anotaciones para el sistema de presentación.

En la literatura sobre NLG Podemos encontrar muchas alternativas en lo que respecta a la especificación de frases. Todas estas varían en el nivel de abstracción que tienen. Las representaciones mas abstractas le darán mas flexibilidad a las etapas de document planning y microplanning, pero al mismo tiempo nos obligarán a tener un realizador de superficie mas sofisticado. Por otro lado, las especificaciones menos abstractas, requieren que el document planner y el microplanner realicen un mayor trabajo, pero también tendrán mas control sobre el texto a producir. Uno de los objetivos que tuvimos a la hora de idear una estructura para nuestra especificación de frases fue que ésta sea independiente de nuestro problema, pretendemos que hable en términos del lenguaje que queremos generar y no en términos específicos de Z en nuestro caso. De esta forma podremos implementar un realizador de superficie que sea independiente de este problema y que pueda ser reutilizado.

Es por esto que decidimos especificar las oraciones a generar mediante árboles sintácticos, donde los constituyentes de éstos serán los sintagmas[1] de la oración. Esto le dará la posibilidad al realizador lingüístico de poder identificar la función de cada uno de los constituyentes de la oración y poder trabajar por por ejemplo con el núcleo de un sintagma particular, debido a que, como vimos en el capítulo 5.2, necesitará conocer el núcleo del sujeto y

[1]Grupo de palabras que ejercen una función sintáctica dentro de una oración

el verbo de una oración para poder asegurarse de que haya concordancia de número y persona entre el verbo y el sujeto.

En la figura 7.2 podemos ver los elementos que utilizamos para modelar las frases de nuestro sistema.

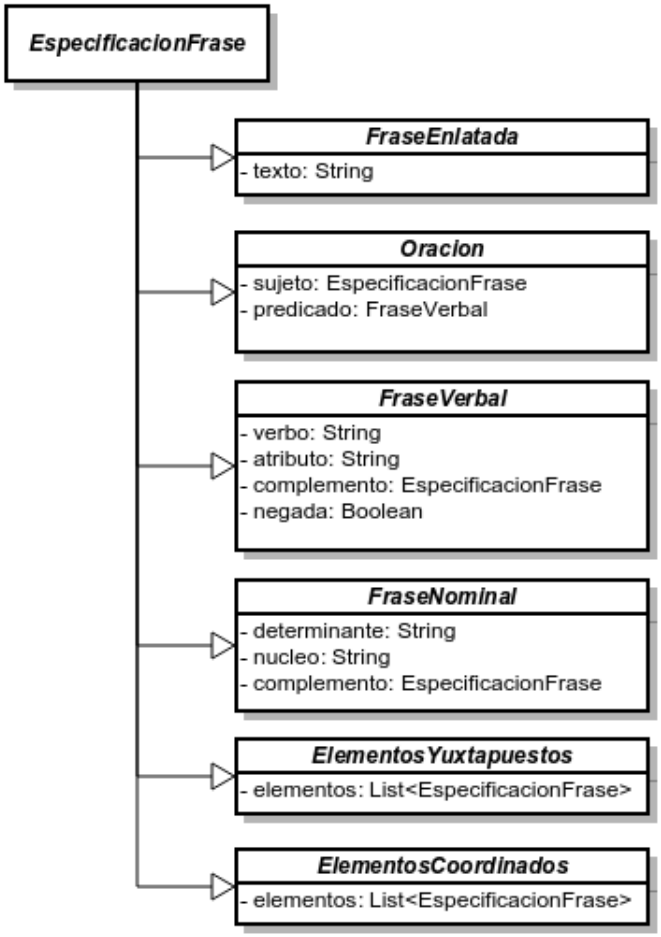

Figura 7.2: Phrase Specification.

Como podemos observar no pretendemos modelar todo el lenguaje sino solo un subconjunto del mismo que provea las herramientas necesarias al realizador para generar las frases definidas en el capítulo 5.2, ya que el desarrollo de un realizador lingüístico que tenga en cuenta todas las construcciones de nuestro lenguaje escapa el alcance de nuestro trabajo.

Es por esto que sólo modelaremos los sintagmas nominales (*FraseNominal*) y verbales (*FraseVerbal*) y nos veremos obligados a incluir otros elementos como *ElementosYuxtapuestos* para salvaguardar la falta de algunos constituyentes sintácticos como sintagmas adjetivales, preposicionales, etc.

A continuación describiremos brevemente cada uno de estos elementos, profundizando mas en detalle sobre la realización de los mismos en el capítulo 8.2.

- ***FraseEnlatada***: Representa texto que no necesita ningún tipo de procesamiento posterior a realizar durante la realización lingüística, será incluido en el texto tal cual fue establecido.
- ***Oracion***: Modela oraciones bimembres. El realizador lingüístico deberá procesarlas en base a una serie de reglas gramaticales para producir un texto sintáctica, morfológica y ortográficamente correcto para éstas.
- ***FraseVerbal***: Representa un sintagma verbal que corresponderá al predicado de una *Oracion*.
- ***FraseNominal***: Modela un sintagma nominal. Generalmente conformará el sujeto en una *Oracion*.
- ***ElementosCoordinados***: Representa una serie de elementos que se deberán transformar en una conjunción de frases en la etapa de realización lingüística, por ejemplo: *"frase1,* ***frase2*** *y frase3"*
- ***ElementosYuxtapuestos***: Representa una lista ordenada de elementos que deberán ser realizados y *concatenados* uno al lado del otro en la oración final. Nos vimos obligados a introducir este tipo de elementos para salvaguardar la falta de algunos constituyentes sintácticos como sintagmas adjetivales, preposicionales, etc.

7.3. Lexicalización

Como mencionamos anteriormente, el proceso de lexicalización será el encargado de elegir que palabras particulares y constructores sintácticos usar para comunicar la información contenida en el document plan. En esta etapa deberemos producir una especificación de frase para cada mensaje contenido en el document plan. En nuestro caso debemos hacerlo utilizando como especificación el conjunto de reglas definidas anteriormente en base al *corpus* en el capítulo 5.2.

El módulo encargado de esta tarea en nuestro sistema deberá ser capaz de generar una especificación de frase a partir de la expresión Z contenida en un mensaje del document plan. Primero deberá verificar si la expresión en cuestión se encuentra designada, en este caso, deberá construir una especificación de frase en base a su designación. De lo contrario deberá intentar construirla recursivamente de acuerdo a las reglas antes mencionadas. En la figura 1 podemos ver un bosquejo del algoritmo del comportamiento que deseamos que tenga nuestro sistema. Este ejemplo es sólo para ilustrar el comportamiento deseado durante esta etapa. El algoritmo real resulta un poco más complejo y deberá construir una *EspecificacionFrase* construyendo y componiendo sintagmas y otros de los elementos definidos en lugar del texto que vemos en el ejemplo.

Algorithm 1 Bosquejo Lexicalización.

```
function LEXICALIZACION(exp)
    if esta_designada(exp) then
        ret ← designacion(exp)
    else
        ret ← lexicalizacion'(exp)
    end if
    return ret
end function

function LEXICALIZACION'({x} = y)
    lexx ← lexicalizacion(x)
    lexy ← lexicalizacion(y)
    return concat(lexx, "es el único elemento de", lexxy)
end function

function LEXICALIZACION'(x = y)
    lexx ← lexicalizacion(x)
    lexy ← lexicalizacion(y)
    return concat(lexx, "es(son) igual(es) a", lexxy)
end function
...
```

La función *designacion* deberá ser capaz de construir una especificación de frase a partir de una expresión designada[2]. Para esto deberá recuperar el texto de esta designación y procesarlo a fin de crear una especificación de frase. Es razonable suponer que el texto de una designación sea un sintagma nominal, es decir tenga la siguiente estructura:

$$\text{Sintagma Nominal} = [\text{Determinante}] + \text{Núcleo} + [\text{Complemento}]$$

Por lo tanto nuestro sistema deberá trabajar el texto de las designaciones (haciendo uso de algún analizador morfológico) para construir una *FraseNominal* a partir de cada designación.

Por otro lado, en el caso que la expresión a lexicalizar no se encuentre designada, se deberá analizar recursivamente la expresión para generar el texto adecuado según las reglas antes mencionadas.

[2]Escribimos esta función de esta forma para simplificar, pero la implementación de la misma es un poco mas compleja en realidad. Además de la expresión deberíamos saber el nombre del esquema al que pertenece la expresión ya que podría tratarse de una variable de esquema por ejemplo. Otro caso particular es el de las designaciones parametrizadas en el que deberá resolverse la designación del argumento primero para luego construir la designación pretendida.

Creemos que describir detalladamente nuestro algoritmo de lexicalización puede resultar muy engorroso para el lector debido a la cantidad de casos que debemos detallar (uno por cara regla de las antes mencionadas) y lo complejas que pueden resultar las especificaciones de frase. Es por esto que ilustraremos el resultado de nuestra lexicalización mediante un pequeño ejemplo.

Tomaremos como ejemplo el primer mensaje del el document plan de la figura 6.4. Nuestra misión será lexicalizar la expresión:

$$s? \in \text{dom}\, st$$

contando con las siguientes designaciones:

$s?$	$\approx$ el símbolo a buscar
$dom\ x$	$\approx$ símbolos cargados en la tabla de símbolos

En una primer instancia, al no encontrarse designada $s? \in \text{dom}\, st$ nuestro lexicalizador deberá construir una *Oracion* para la frase:

$$\text{lexicalizacion}\ s? +\!+ \text{“pertenece a”} +\!+ \text{lexicalizacion dom}\, st$$

Para esto deberá resolver recursivamente la lexicalizacion de $s?$ y de dom st para formar el *sujeto* y *predicado* necesarios para la oración. Luego deberá formar una *FraseVerbal* completando tanto el *verbo* como el *complemento* a fin de obtener el texto “pertenece a” en el texto final. En la figura 7.3 podemos observar el resultado final de la lexicalización.

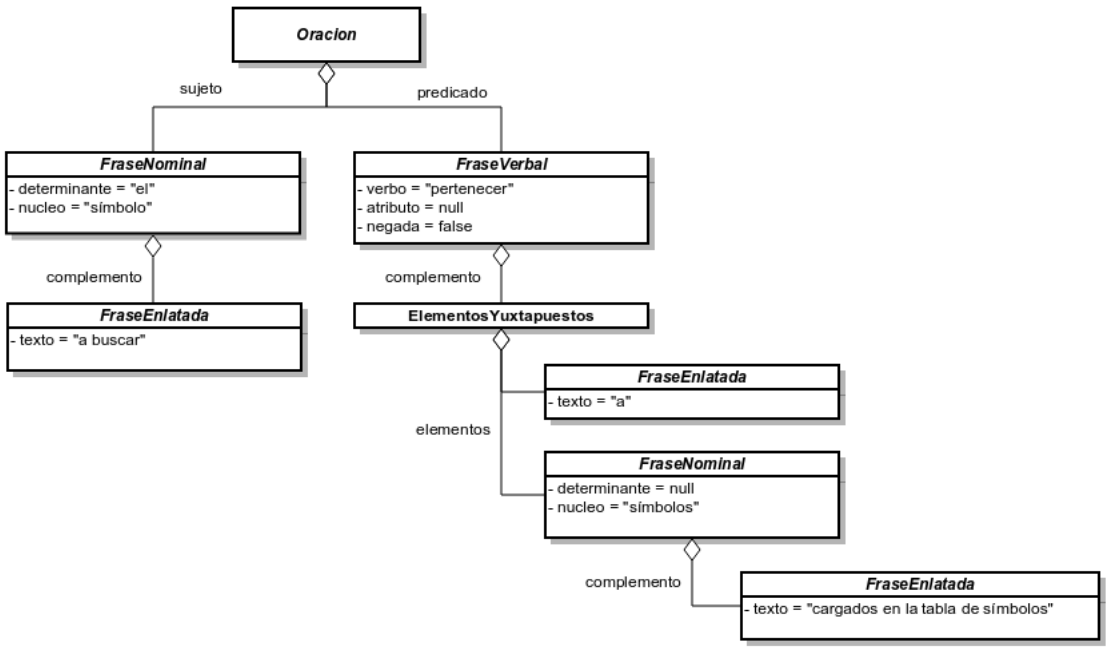

Figura 7.3: Phrase Specification para.

Cabe aclarar que estableceremos el verbo en infinitivo siendo luego el realizador lingüístico el encargado de conjugar el mismo de acuerdo a algunas reglas gramaticales que veremos en el capítulo 8.2. Otra cuestión a mencionar es el uso del elemento *ElementoYuxtapuestos* para salvaguardar la falta de un elemento que nos sirva para modelar un sintagma preposicional en este caso. Nuestro realizador lingüístico procesará los elementos contenidos en cada *ElementoYuxtapuestos* generando un texto resultado de la concatenación de la realización los mismos.

Capítulo 8

Realización de superficie

En los capítulos anteriores vimos los procesos que necesitan ser llevados a cabo para lograr una especificación del texto a generar. Finalmente, en esta última etapa de nuestro sistema nos concentraremos en estudiar las tareas que deben ser realizadas a fin de transformar esta especificación del texto en texto de superficie, formado por frases, símbolos de puntuación y algunas etiquetas de mark-up necesarias.

Como mencionamos en el capítulo 7, nuestra especificación del texto es un árbol en el que las horas representan las frases individuales y los nodos internos establecen como éstas están agrupadas (según estructuras como párrafos, secciones, listas de ítems, etc). En base a esta diferenciación entre los elementos de nuestra especificación del texto podemos distinguir dos grandes aspectos dentro de esta tarea: la *realización estructural* responsable de transformar los nodos internos de nuestra especificación de texto en anotaciones particulares del sistema de presentación y por otro lado, la tarea de *realización lingüística* que se concentrará en generar frases sintáctica, morfológica, ortográficamente correctas.

8.1. Realización estructural

Como mencionamos anteriormente, en la etapa de realización estructural se deberán transformar los constructores lógicos existentes en la especificación del texto en constructores del sistema de presentación. Hoy en día no debemos ocuparnos de cuestiones de formateo de bajo nivel, sino que la mayoría de los sistemas de procesamiento de texto nos permiten indicar mediante el uso de símbolos o etiquetas la naturaleza de una estructura determinada; estas luego estas serán procesadas y renderizadas de manera apropiada permitiéndole al lector una correcta visualización.

Es evidente que esta etapa será dependiente del sistema de presentación escogido para el documento final y a su vez independiente del proceso de realización lingüística de la las oraciones o frases a generar. Por ejemplo, podemos

observar en la figura 8.1 el código LaTeXpara una lista de items de una descripción y en la figura 8.2 la realización para la misma especificación de texto pero en lenguaje HTML, donde el texto contenido en ambos ejemplos es exactamente el mismo, sólo diferencian en las diferentes anotaciones utilizadas para cada sistema de presentación.

```
LookUp\_SP\_1: Se busca un símbolo en la tabla.
\begin{itemize}
  \item{Cuando:}
  \begin{itemize}
    \item{El símbolo a buscar pertenece ...}
    \item{El símbolo a buscar es el único ...}
  \end{itemize}
\end{itemize}
```

Figura 8.1: Texto final en LaTeX.

```
LookUp_SP_1: Se busca un símbolo en la tabla.
<ul>
  <li>Cuando:</li>
  <ul>
    <li>El símbolo a buscar pertenece ...</li>
    <li>El símbolo a buscar es el único ...</li>
  </ul>
</ul>
```

Figura 8.2: Texto final en HTML.

Podemos pensar el problema de realización estructural como el proceso de mapear los constructores lógicos de nuestra especificación del texto en constructores lógicos del lenguaje de presentación a utilizar. En nuestro caso sólo tenemos que considerar dos elementos: *TSDocumento* y *TSListaItems*.

Para realizar *TSDocumento* se deberán agregar algunas etiquetas de encabezado necesarias según el tipo del documento, deberemos especificar el título y generalmente deberemos delimitar el principio y final del texto a incluir en el documento. Luego deberemos realizar recursivamente los elementos contenidos en el documento (*TSListaItems*) y ubicarlos adecuadamente en el cuerpo del mismo. En la figura 8.3 podemos ver un ejemplo del código que deberíamos generar para la realización estructural del documento, la realización de *TSListaItems* se puede apreciar en el ejemplo anterior de la figura 8.1.

```
\documentclass{article}
\title{titulo}

\begin{document}
\maketitle
    ...
\end{document}
```

Figura 8.3: Código LaTeXpara estructura del documento.

8.2. Realización lingüística

La realización lingüística desempeñará su tarea al nivel de la oración. Como mencionamos anteriormente, la misión de esta tarea será transformar las especificaciones de frases de nuestro sistema en oraciones bien formadas. Entendiendo por oración bien formada aquellas que cumplan con las reglas gramaticales del lenguaje (español en nuestro caso); estas reglas se ocuparán tanto de la morfología como de la sintaxis de las mismas. Entonces, la realización lingüística consistirá en aplicar alguna caracterización de estas reglas a cada especificación de frase a fin de producir un texto que sea sintáctica y morfológicamente correcto.

Es importante también, que el texto producido sea ortográficamente correcto. Para esto, asumiremos que las designaciones provistas por el usuario son ortogáficamente correctas, no tendremos que comprometer a que las palabras generadas por nuestro sistema también lo sean y finalmente nos deberemos encargar de que la primer palabra de una oración comience con mayúscula y de colocar un signo de puntuación al final de la misma.

En base a las reglas introducidas en el capítulo 5.2 podemos extraer algunos de los aspectos sintácticos y morfológicos que debemos tener en cuenta durante esta etapa para ser capaces de producir los textos esperados. En particular nos focalizaremos en tres reglas de *concordancia gramatical* que nuestro realizador lingüístico deberá contemplar, extraídas del "Diccionario panhispánico de dudas" de la Real Academia Española [**?**].

Concordancia entre artículo y sustantivo. Establece que el artículo debe concordar en género y número con el sustantivo al que acompaña. Como mencionamos anteriormente, el usuario podría no incluir el artículo en una designación. Por ejemplo, continuando con el ejemplo de la figura 3, se podría haber omitido el artículo de "el símbolo a buscar" y solo haber designado:

$$s? \approx \text{símbolo a buscar}$$

Será entonces nuestro sistema el encargado de agregar el artículo de ser necesario[1], asegurándose de que el mismo concuerde con el sustantivo. Para esto deberemos ser capaces de identificar el género y número del sustantivo mediante el uso de un analizador morfológico.

Concordancia entre sujeto y atributo de verbo copulativo. Establece que el atributo de un verbo copulativo (ser, estar, parecer) debe concordar en género y número con el sujeto. Veamos por ejemplo la regla para describir el operador $\subset$ definida en el capítulo 5.2

$exp1 \subset exp2$
$\rightarrow$ `verb(exp1)` + "*está(n) incluido/a(s) en*" + `verb(exp2)`

Como podemos ver, tenemos el verbo copulativo *"está"* acompañado del atributo *"incluido"*. En este caso deberemos escojer el genero y numero del atributo de forma tal que concuerde con el sujeto.

Concordancia entre sujeto y verbo. El verbo debe concordar con el sujeto en número y persona. En el caso de haber varios sujetos, la concordancia debe hacerse con el verbo en plural. Observemos por ejemplo la regla por defecto para el operador $\subset$:

$\{exp1, exp2, \ldots, expn\} \subset expm$
$\rightarrow$`verb(exp1)` + "," + `verb(exp2)` + "*, ..., y*" + `verb(expn)` + "*pertenece(n) a*" + `verb(expm)`

Si la expresión a la izquierda del $\subset$ es un conjunto unitario deberemos usar el verbo *"pertenece"* (en singular), pero de tratarse de un conjunto con más de un elemento, deberemos usar el verbo en plural.

Otra cuestión a observar, además de las reglas de concordancia ya vistas, es que nuestro sistema deberá generar casi exclusivamente oraciones bimembres ordenadas como *sujeto + verbo + objeto*. Siempre expresadas en tiempo presente. Por lo tanto nuestro realizador siempre deberá respetar el orden de palabras y tiempo verbal mencionado para realizar los elementos de tipo *Oracion* de nuestra especificación de frase.

Debemos aclarar que en las reglas mencionadas anteriormente podemos observar oraciones en las cuales el orden de palabras difiere del mencionado anteriormente, por ejemplo: la frase "no hay elementos en ..." cuyo orden es *verbo + sujeto + objeto*. Para poder generar estas frases no modelaremos estos casos como una *Oracion* sino como *ElementosYuxtapuestos* entre una *FraseEnlatada* con el texto "no hay elementos en" y el objeto al que se haga referencia. Podemos hacerlo de esta forma ya que no tenemos necesidad procesar la especificación de la frase "no hay elementos en" ya que sabemos de

[1] Podría no ser necesario, por ejemplo si el núcleo de la frase designada fuese un nombre propio.

antemano que es sintáctica y morfológicamente correcta. Esto no podríamos asegurarlo si el sujeto de la oración pudiese ser una designación por ejemplo, pero en este caso, ya lo conocemos.

En base a los aspectos mencionados anteriormente analizaremos la tarea de realización lingüística para cada elemento de nuestra especificación de frase ilustrando cada caso con un pseudocódigo para el algoritmo de verbalización. Llamaremos *verbalizar* a la función encargada de generar una oración sintáctica y mofológicamente correcta en lenguaje natural en base a una especificación de frase. Ésta resultará la función principal de nuestro realizador lingüístico dando como resultado una oración a la que sólo deberemos realizarle algunas pequeñas modificaciones para satisfacer los aspectos ortográficos antes mencionados y dar por finalizada la tarea de realización lingüística.

FraseEnlatada. La realización de una frase enlatada será trivial, simplemente habrá que extraer el texto contenido en la misma sin necesidad de realizar ningún tipo de procesamiento.

Algorithm 2 Realización lingüística frase enlatada.

```
1: function VERBALIZAR(FraseEnlatada frase)
2:     return frase.texto
3: end function
```

ElementosYuxtapuestos. Representa una concatenación de frases. El resultado de la verbalización deberá ser un texto que resulte de la unión de las verbalizaciones individuales de cada uno de los elementos contenidos, agregando los espacios correspondientes entre estos[2] y respetando el orden en que se encuentren.

Algorithm 3 Realización lingüística elementos yuxtapuestos.

```
1: function VERBALIZAR(ElementosYuxtapuestos elem)
2:     for each e in elem.elementos do
3:         resultado ← concat(resultado, verbalizar(e))
4:     end for
5:     return resultado
6: end function
```

ElementosCoordinados. Se trata de una serie de elementos que deberán ser verbalizados individualmente y unidos, de una forma similar a la anterior, a fin de obtener una conjunción de frases.

[2]Suponemos que la función *concat* además de concatenar los elementos agrega un espacio entre cada uno de éstos.

Algorithm 4 Realización lingüística elementos yuxtapuestos.

```
1: function VERBALIZAR(ElementosCoordinados elem)
2:     for each e in elem.elementos do
3:         if esPrimerElemento(e, elem) then
4:             resultado ← verbalizar(e)
5:         else if esUltimoElemento(e, elem) then
6:             resultado ← concat(resultado, "y", verbalizar(e))
7:         else
8:             resultado ← concat(resultado, ",", verbalizar(e))
9:         end if
10:    end for
11:    return resultado
12: end function
```

FraseNominal. Para verbalizar una frase nominal, deberemos unir en orden: *determinante* (de ser requerido), *nucleo* y la verbalización del *complemento*, agregando los espacios correspondientes entre medio.

Algorithm 5 Realización lingüística FraseNominal.

```
1: function VERBALIZAR(FraseNominal frase)
2:     nucleo ← frase.nucleo
3:     complemento ← verbalizar(frase.complemento)
4:     if esNombre(nucleo) then
5:         determinante ← determinar_articulo(frase)
6:         resultado ← concat(determinante, nucleo, complemento)
7:     else
8:         resultado ← concat(nucleo, complemento)
9:     end if

10:    return resultado
11: end function
```

La función *determinar_articulo* deberá recuperar el determinate apropiado para la frase, en el caso de encontrarse ya establecido de antemano devolverá el mismo, en caso contrario (de ser necesario) deberá determinar el artículo indicado según número y género del núcleo de la frase. Observemos que se deberá verbalizar recursivamente el complemento de la frase (éste probablemente sea una frase enlatada o una yuxtaposición de éstas). Finalmente se construye un texto con el *determinante*, *núcleo* y la verbalización del *complemento*, en este orden.

FraseVerbal. No realizaremos un análisis individual de este caso ya que la realización del mismo dependerá de otros elementos dentro de la oración. Al

verbalizar un elemento de tipo *Oracion* analizaremos la *FraseVerbal* que corresponde al predicado del mismo.

Oracion. Este es el caso mas interesante para nuestro verbalizador, deberemos generar una oración correcta en base a las reglas antes vistas.

Algorithm 6 Realización lingüística Oracion.

```
1:  function VERBALIZAR(Oracion oracion)
2:      sujeto ← verbalizar(oracion.sujeto)
3:      verbo ← determinar_verbo(oracion)
4:      atributo ← determinar_atributo(oracion)
5:      complemento ← verbalizar(oracion.predicado.complemento)
6:      estaNegada ← oracion.predicado.negada

7:      if estaNegada then
8:          resultado ← concat(sujeto, negacion(verbo), atributo, complemento)
9:      else
10:         resultado ← concat(sujeto, verbo, atributo, complemento)
11:     end if

12:     return resultado
13: end function
```

A fin de contemplar las reglas de concordancia introducidas anteriormente, le prestaremos especial atención al verbo y al atributo de la frase verbal corresponde predicado de la oración, verbalizando individualmente tanto el sujeto (que probablemente sea una frase nominal) como el complemento de la frase verbal. La función *determinar_verbo* será la encargada de conjugar el verbo de manera que concuerde en número y persona con el sujeto, mientras que la función *determinar_atributo* será la encargada de determinar el atributo de forma que concuerde en número y género también con el sujeto. Si se trata de una negación (por ejemplo *"... no pertenece a ..."*) deberemos negar el verbo en cuestión en la oración final. Finalmente se unirán los constituyentes en el orden mencionado previamente: *sujeto - verbo - predicado.*

Apéndice A

Corpus de descripciones

...

Bibliografía

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[BCCM99] A. Bertani, W. Castelnovo, E. Ciapessoni, and G. Mauri. Natural language translations of formal specifications for complex industrial systems. In *Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence*, pages 185–194, 1999.

[CAF+11] Maximiliano Cristiá, Pablo Albertengo, Claudia Frydman, Brian Pluss, and Pablo Rodriguez Monetti. Applying the test template framework to aerospace software. In *Proceedings of the 2011 IEEE 34th Software Engineering Workshop*, SEW '11, pages 128–137, Washington, DC, USA, 2011. IEEE Computer Society.

[CBB+97] R. G. G. Cattell, Douglas K. Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[CM09] Maximiliano Cristiá and Pablo Rodríguez Monetti. Implementing and applying the stocks-carrington framework for model-based testing. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 167–185, Berlin, Heidelberg, 2009. Springer-Verlag.

[Cos97] Yann Coscoy. A natural language explanation for formal proofs. In *Selected Papers from the First International Conference on Logical Aspects of Computational Linguistics*, LACL '96, pages 149–167, London, UK, UK, 1997. Springer-Verlag.

[CP10] Maximiliano Cristiá and Brian Plüss. Generating natural language descriptions of z test cases. In *Proceedings of the 6th International*

Natural Language Generation Conference, INLG '10, pages 173–177, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[GMM90] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *J. Syst. Softw.*, 12(2):107–123, May 1990.

[Jac95] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[LRR97] Benoit Lavoie, Owen Rambow, and Ehud Reiter. Customizable descriptions of object-oriented models. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, ANLC '97, pages 253–256, Stroudsburg, PA, USA, 1997. Association for Computational Linguistics.

[PTSF97] J. M. Punshon, J. P. Tremblay, P. G. Sorenson, and P. S. Findeisen. From formal specifications to natural language: A case study. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 309–, Washington, DC, USA, 1997. IEEE Computer Society.

[RD00] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*. Cambridge University Press, New York, NY, USA, 2000.

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[SSTP94] A. Salek, P. G. Sorenson, J. P. Tremblay, and J. M. Punshon. The review system: From formal specifications to natural language. In *Proceedings of the First International Conference on Requirements Engineering*, pages 220–229, 1994.

[STM88] Paul G. Sorenson, Jean-Paul Tremblay, and Andrew J. McAllister. The metaview system for many specification environments. *IEEE Softw.*, 5(2):30–38, March 1988.