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

Directores
Maximiliano Cristiá
Brian Plüss

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

Noviembre 2015

Índice general

Índice de figuras

Índice de Algoritmos

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 a testear, y a partir del mismo son generados los casos de prueba.

Un caso particular del testing basado en modelos es el *Test Template Framework* (abreviado **TTF**) descrito por Stocks y Carrington [SC96]. El TTF utiliza como modelo de entrada una especificación formal escrita en notación *Z* [Spi92] y establece cómo generar *casos de prueba* para las operaciones incluidas en la especificación.

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 llamados *clases de prueba* y luego, a partir de los mismos, generar los *casos de prueba concretos*.

Por otro lado, 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 para leer los casos de prueba (generados mediante el TTF, por ejemplo) y comprender lo que está siendo testeado con los mismos. 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 1.1 podemos observar, a modo ilustrativo, un caso de prueba (generado mediante el TTF) para la operación *Update* perteneciente a una especificación para una tabla de símbolos y luego, en la figura 1.2, una posible descripción en lenguaje natural del mismo. Una tabla de símbolos es una estructura de datos creada y mantenida por un compilador con el fin de almacenar información (ubicación, ámbito, etc.) relativa a los distintos

elemento del código fuente como: variables, nombres de funciones, objetos, etc. Ésta provee al menos dos operaciones: búsqueda e inserción (*Update* en nuestro caso será la encargada de insertar y actualizar símbolos en la tabla). En la sección 2.1.1 nos explayaremos más en detalle sobre el funcionamiento de una tabla de símbolos y presentaremos la especificación Z utilizada para el ejemplo.

Update_SP_4_TCASE

Update_SP_4

$st = \{(sym0, val0)\}$
$s? = sym0$
$v? = val0$

Figura 1.1: Caso de prueba para operación *Update*

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 1.2: Posible descripción en lenguaje natural para *Update_SP_4*

Contar con una descripción en lenguaje natural como la de la figura 1.2, sería de gran ayuda para que la persona a cargo de la validación y verificación comprenda lo que está siendo testeado y además, fundamentalmente, para conectar la especificación del caso de prueba con lo que esto significa en la implementación.

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 del testing.

El objetivo de este trabajo, será entonces, desarrollar una solución para la generación automática de descripciones para los casos de prueba generados por el TTF. Para esto, utilizaremos técnicas de *generación de lenguaje natural* (abreviado **NLG** del inglés *"natural language generation"*). En particular, seguiremos la metodología más comúnmente aceptada para la construcción de sistemas de NLG, propuesta por Reiter y Dale [RD00]. Trabajaremos principalmente con las *clases de prueba*, ya que estas son las que contienen la información referente a las alternativas funcionales que se intentan testear

mediante cada *caso de prueba* generado. Además deseamos idear una solución independiente del número de operaciones del modelo así como del dominio de aplicación del mismo. Para esto, junto con la utilización de técnicas de NLG, haremos uso de la información contenida en las designaciones [Jac95] (asociaciones entre los elementos de la especificación y elementos que refieren al dominio de la aplicación) que deberían acompañar la especificación formal.

Como resultado de este trabajo también se realizará la implementación de un prototipo, a desarrollarse en lenguaje Java e integrarse 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 meta-sistema que facilita la construcción de entornos CASE (*Computer Aided Software Engineering*) 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 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] un método 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 presenta una solución ad-hoc basada en *templates*, donde los *templates* utilizados son dependientes del dominio de aplicación y de la cantidad de operaciones en la especificación. Basado en este trabajo, 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] http://www.fceia.unr.edu.ar/~mcristia/fastest-1.6.tar.gz

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 1.3 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. *mod*

Figura 1.3: Expresiones soportadas

1.4. Estructura de la tesina

En este capítulo presentamos los objetivos principales para este trabajo. Establecimos como uno de estos lograr una solución superadora al trabajo realizado por Cristiá y Plüss [CP10] (único hasta el el momento en trabajar sobre generación de lenguaje natural para Z y el TTF). Puntualmente, a diferencia del anterior, nuestro objetivo será lograr una generación de descripciones en

lenguaje natural independiente del dominio de aplicación y de la cantidad de clases y casos de pruebas.

En el próximo capítulo introduciremos conceptos básicos acerca del *test template framework* fundamentales para el desarrollo de este trabajo. Luego, en el capítulo 3, presentaremos la metodología a utilizar para diseñar y construir nuestro sistema de NLG. En el capítulo 4 realizaremos un análisis de requerimientos en base al corpus de datos recolectado, que resultará de vital importancia para el desarrollo de las distintas tareas que deberá realizar nuestro sistema - las cuales estudiaremos en profundidad en los capítulos 5, 6 y 7. En el capítulo 8 veremos los aspectos más relevantes de la implementación realizada y finalmente en el capítulo 9 presentaremos las conclusiones de esta tesina, así como también posibles trabajos futuros.

Capítulo 2

Fundamentos del *testing* basado en modelos

En este capítulo introduciremos algunos conceptos básicos con los que trabajaremos a lo largo de esta tesina. Presentaremos el *testing* basado en modelos y veremos mediante un ejemplo como derivar clases y casos de prueba a partir de una especificación Z. Además, presentaremos el funcionamiento de Fastest, la herramienta utilizada para generar clases y casos de prueba de manera automática, en la que basaremos todo el desarrollo a realizar como parte de este trabajo. Finalmente estudiaremos el importante rol que cumplirán las designaciones nuestro sistema de NLG, fundamental para la generación de textos independientes del dominio de aplicación.

2.1. *Testing* basado en modelos

La hipótesis fundamental detrás del *testing* basado en modelos (MBT) es que un programa es correcto si verifica su especificación, entonces, la especificación resulta una excelente fuente para obtener casos de prueba. Las técnicas de MBT utilizan la especificación, en primer instancia, para derivar casos de pruebas abstractos (al nivel del modelo). Estos luego deben ser refinados al nivel del lenguaje de implementación y ejecutados por el programa que supuestamente implementa la especificación. Finalmente la salida del programa será abstraída al nivel de la especificación y la misma será utilizada (nuevamente) para verificar si el caso de prueba ha detectado un error. En el esquema de la figura 2.1 podemos ver el proceso recién mencionado.

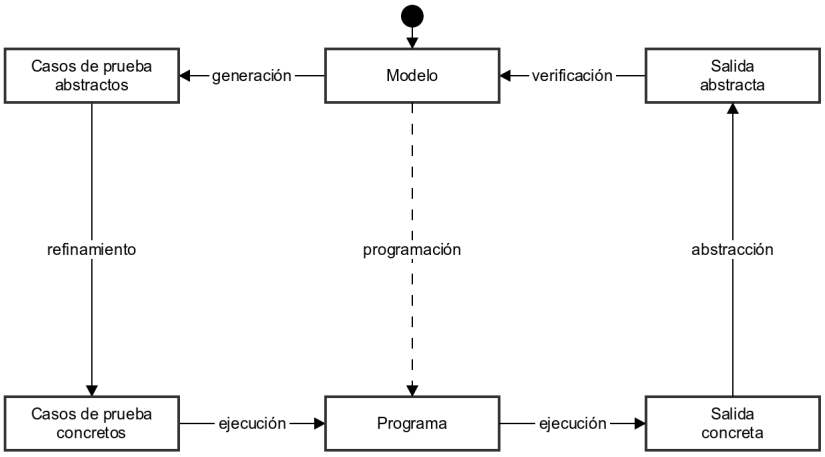

Figura 2.1: Proceso de *testing* basado en modelos

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

A continuación introduciremos brevemente el TTF mediante un ejemplo, asumiendo que el lector se encuentra familiarizado con la notación Z[1].

2.1.1. Ejemplo: *Symbol Table*

Una tabla de símbolos es una estructura de datos utilizada por un compilador o intérprete 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 correspondiente a un símbolo ya cargado en la misma.

Tipos elementales. Es irrelevante para nuestro modelo especificar en detalle la estructura con la cual representaremos el conjunto de símbolos aceptados por el compilador/interprete, así como la información relativa a los mismos. Para esto, podemos abstraer los conjuntos antes mencionados utilizando los siguientes tipos básicos:

[1]Introducción a la notación Z: http://www.fceia.unr.edu.ar/asist/z-a.pdf

$[SYM, VAL]$

El tipo básico SYM representará el conjunto de todos los símbolos aceptados por el compilador/interprete, mientras que VAL abstraerá el conjunto de toda la información que pudiese estar asociada a un símbolo.

Estado de la tabla. Es natural pensar que la tabla de símbolos establece una relación funcional entre los símbolos aceptados y la información asociada a cada uno de ellos. Por otro lado, el estado de nuestra tabla está formado únicamente por los símbolos cargados y la información de cada uno. Por lo tanto, podemos modelar el conjuntos de estados de la tabla mediante una única función parcial[2] de la siguiente manera:

ST

$$st : SYM \pfun VAL$$

Operaciones. Como mencionamos anteriormente, dos operaciones se realizarán sobre la tabla de símbolos: inserción y búsqueda.

Comenzaremos por modelar la operación de inserción o actualización. Esta operación modificará el estado de la tabla de símbolos; para esto, será necesario brindarle a la operación el símbolo y la información correspondiente al mismo.

Para actualizar la información de un símbolo en la tabla, simplemente actualizaremos la relación funcional con un par ordenado construido a partir del símbolo y la información dada.

Update

$$\Delta ST$$
$$s? : SYM$$
$$v? : VAL$$
$$rep! : REPORT$$

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

Por último, deberemos especificar la operación encargada de recuperar la información vinculada a un símbolo. Se trata de una operación relativamente

[2] Utilizamos una función parcial ya que no todos los símbolos aceptados por el compilador estarán presentes en la tabla.

simple que no modifica el estado de la tabla de símbolos. Ésta requerirá de un símbolo a buscar y retornará la información vinculada al mismo. A continuación especificaremos el caso exitoso para esta operación:

LookUpOk

$$\Xi ST$$
$$s? : SYM$$
$$v! : VAL$$
$$rep! : REPORT$$

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

Utilizamos la variable de salida *rep!* para para comunicar el éxito o fracaso de la operación, y la variable *v!* para retornar la información solicitada.

El esquema *LookUpOk* modela solamente el caso en el que el símbolo a buscar se encuentre cargado en la tabla, pero deberemos contemplar también el caso en que se intente buscar un símbolo que no haya sido cargado en la misma. Modelaremos esta situación con el esquema de error *LookUpE*, presentado a continuación:

LookUpE

$$\Xi ST$$
$$s? : SYM$$
$$rep! : REPORT$$

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

Finalmente, la operación total (que es la que se debe programar) queda definida de la siguiente manera:

$$LookUp == LookUpOk \lor LookUpE$$

2.1.2. *Test Template Framework*

La propuesta de Stocks y Carrington [SC96] es utilizar la especificación Z de una operación como fuente de la cual obtener casos de prueba 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 estas alternativas funcionales.

2.1.3. Clases y casos de prueba

Las alternativas funcionales antes mencionadas pueden expresarse como restricciones sobre las variables de entrada y de estado definidas para la especificación. Estas alternativas pueden ser especificadas mediante esquemas Z a los que llamaremos *clases de prueba*. Por ejemplo:

$$LookUp_1^{FND} == [st : SYM \pfun VAL;\ s? : SYM \mid s? \in \mathrm{dom}\, st]$$

define una clase de prueba para el esquema *LookUp*, introducido anteriormente, donde st y $s?$ son variables de estado y de entrada respectivamente, y $s? \in \mathrm{dom}\, st$ es la restricción sobre las mismas.

Por otro lado, necesitaremos buscar valores específicos, o constantes, para las variables de una *clase de prueba* (que reduzcan su restricción a verdadero) a fin de luego refinarla y poder ejecutar el caso de prueba en el programa que implemente la especificación. Llamaremos *caso de prueba* a una tupla de valores para las variables involucradas en la clase de prueba que cumplan con la restricción de la misma. Nos bastará con escoger un único *caso de prueba* para cada clase de prueba generada (observemos que en algunos casos, como por ejemplo para la clase de prueba antes presentada, podríamos encontrar infinitos valores posibles para las variables de la clase de prueba que cumplan con la restricción).

Por otro lado, en muchos casos, necesitaremos definir las constantes necesarias para los tipos involucrados en la *clase de prueba*, por ejemplo, para los tipos SYM y VAL involucrados en la clase de prueba presentada anteriormente, podríamos definir:

$$\begin{array}{|l} sym_1 : SYM \\ val_1 : VAL \end{array}$$

Finalmente, estamos en condiciones de definir un *caso de prueba* correspondiente a la clase de prueba $LookUp_1^{FND}$ de la siguiente manera:

$$LookUp_{TC}^1 == [LookUp_1^{FND} \mid st = \{sym_1 \mapsto val_1\} \land s? = sym_1]$$

2.1.4. Generación de clases de prueba

En esta sección introduciremos brevemente, por medio de un ejemplo, el proceso del TTF. Continuaremos trabajando sobre el ejemplo introducido anteriormente e intentaremos generar algunos casos de prueba para testear la operación *LookUp*. En particular nos concentraremos en la generación de clases de prueba, que resultarán de vital interés para nuestro trabajo de NLG.

El primer paso del TTF será definir el espacio de entrada (*IS*, por *Input Space*) para la operación. Este será el conjunto definido por todos los posibles valores de entrada y estado de la misma. Por ejemplo, el *IS* para la operación *LookUp* será:

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

En los casos en los que las operaciones son parciales, no tendrá sentido probar el sistema con casos de prueba para los cuales no está definida la operación, es por esto que, a partir del *IS*, el TTF definirá luego el espacio válido de entrada (*VIS*, por *Valid Input Space*). Este será un subconjunto del anterior, formado por los elementos pertenecientes al *IS* que cumplan con la precondición de la operación en cuestión, es decir:

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

El *VIS* será un subconjunto del *IS* para los cuales tiene sentido testear el programa. En el caso de *LookUp*, su *VIS* es igual a su *IS*, ya que la operación es total. El *IS* y el *VIS* no coincidirían si la operación *LookUp*, por ejemplo, no se hubiera incluido el esquema de error. En ese caso tendríamos:

$$IS == [st : SYM \pfun VAL;\ s? : SYM]$$
$$VIS_{LookUpOk} == [IS \mid s? \in \mathrm{dom}\ st]$$

Luego de determinar el *VIS*, el TTF propone dividir el mismo, de modo tal que cada una de estas particiones represente una alternativa funcional distinta de la operación a testear. Estas particiones serán las *clases de prueba*, introducidas anteriormente, y serán el resultado de aplicar distintas tácticas de *testing* sobre el *VIS*. Luego, será posible aplicar nuevas tácticas sobre estas particiones generadas a fin de dividir nuevamente las mismas, obteniendo como resultado nuevas clases de prueba; 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 será la selección de al menos un *caso de prueba* para cada clase de prueba, que, como vimos anteriormente, consistirá en buscar valores para las variables de la misma que reduzcan su restricción a verdadero.

A continuación, mostraremos la aplicación de algunas tácticas de *testing* sobre la operación *LookUp*. En primer lugar aplicaremos Forma Normal Disyuntiva (DNF del inglés *"Disjunctive Normal Form"*) y luego aplicaremos la táctica de Partición Estándar (SP del inglés *"Standard Partition"*) a la expresión: "$s? \in \mathrm{dom}\ st$".

Forma Normal Disyuntiva. Suele ser la primer táctica que se aplica. Esta expresará la operación como una disyunción de esquemas en los cuales únicamente habrá conjunciones de literales o de negaciones de literales y luego dividirá el *VIS* con las pre-condiciones de cada esquema.

Continuando con el ejemplo, *LookUp* ya es una disyunción y cada uno de los esquemas que la forman se encuentra en DNF. Por lo tanto, solo deberemos dividir el *VIS* con la precondición de cada uno de ellos, de la siguiente forma:

$$LookUp_1^{DNF} == [VIS_{LookUp} \mid s? \in \text{dom } st]$$
$$LookUp_2^{DNF} == [VIS_{LookUp} \mid s? \notin \text{dom } st]$$

De esta forma obtenemos una partición del *VIS*[3]. Además, observemos que si tomamos un caso de prueba para cada una de las clases obtenidas estaríamos probando el sistema en las siguientes situaciones:

1. Intentar buscar un símbolo cargado en la tabla de símbolos.
2. Intentar buscar un símbolo que no fue cargado previamente en la tabla de símbolos.

Partición Estándar. Esta táctica trata con los operadores matemáticos de una operación. Una partición estándar es una partición del dominio del operador en conjuntos llamados *sub-dominios*. En la figura 2.2 podemos ver una partición estándar para los operadores $\in$ y $\notin$.

1. $A = \{\}$	3. $A = \{b\}$	5. $A = \{b, c\}$	7. $\{b, c\} \subset A, a \notin A$
2. $A = \{a\}$	4. $A = \{a, b\}$	6. $\{a, b\} \subset A$	

Figura 2.2: Partición estándar para a $\in$ A; se asume que a, b y c son tres elementos diferentes

Para aplicar esta táctica primero hay que seleccionar un operador de un predicado incluido en el esquema de la operación a testear.

Siguiendo con el ejemplo anterior, aplicaremos la técnica de partición estándar a la clase de prueba *LookUp_DNF_1*. En primer lugar, deberemos seleccionar una aparición de un operador en el esquema de la operación *LookUp*. Nosotros elegiremos el operador $\in$ de la expresión "$s? \in \text{dom } st$".

El siguiente paso será reemplazar los parámetros formales que aparecen en la descripción de la partición por las expresiones usadas en la especificación. En particular, para nuestro ejemplo, deberemos reemplazar los parámetros de las expresiones que aparecen en la figura 2.2 con las expresiones "$s?$" y

[3]Esto no se cumple en todos los casos, podrían "solaparse" las particiones obtenidas, pero de cualquier forma lograríamos un cubrimiento funcional básico.

"dom st". Finalmente, a partir de la clase de prueba sobre la cual se quiere aplicar la táctica, tendremos que generar las nuevas particiones para cada uno de los sub-dominios definidos en la partición estándar. En consecuencia, obtendremos las siguientes clases:

$$
\begin{aligned}
LookUp_1^{SP} &== [LookUp_1^{DNF} \mid \mathrm{dom}\ st = \{\}] \\
LookUp_2^{SP} &== [LookUp_1^{DNF} \mid \mathrm{dom}\ st = \{s?\}] \\
LookUp_3^{SP} &== [LookUp_1^{DNF} \mid \mathrm{dom}\ st = \{b\}] \\
LookUp_4^{SP} &== [LookUp_1^{DNF} \mid \mathrm{dom}\ st = \{s?, b\}] \\
LookUp_5^{SP} &== [LookUp_1^{DNF} \mid \mathrm{dom}\ st = \{b, c\}] \\
LookUp_6^{SP} &== [LookUp_1^{DNF} \mid \{s?, b\} \subset \mathrm{dom}\ st] \\
LookUp_7^{SP} &== [LookUp_1^{DNF} \mid \{b, c\} \subset \mathrm{dom}\ st \land s? \notin \mathrm{dom}\ st]
\end{aligned}
$$

Luego podríamos continuar aplicando nuevas tácticas y particionando aún más el *VIS*. En este caso se trata de una operación relativamente sencilla y consideramos que todas las alternativas funcionales importantes se encuentran representadas mediante las clases de prueba obtenidas.

2.1.5. Fastest

Fastest[4] 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 máximo posible, el proceso de *testing* funcional basado en especificaciones Z. *Fastest* se encuentra implementado mayormente en lenguaje Java y hace uso de las librerías del *framework* CZT[5] (*Community Z Tools*) para contar con utilidades relacionadas al lenguaje de especificación Z.

A continuación, ilustraremos el funcionamiento de *Fastest* mostrando los comandos necesarios para generar las clases de prueba de la sección anterior[6].

[4] http://www.fceia.unr.edu.ar/~mcristia/fastest-1.6.tar.gz

[5] http://czt.sourceforge.net/

[6] Puede ser necesario modificar la partición estándar para el operador $\in$ utilizada por *Fastest* para obtener los mismos resultados ya que la versión 1.6 de *Fastest* utiliza una Partición Estándar diferente para el operador $\in$.

```
Fastest version 1.6, (C) 2013, Maximiliano Cristiá
Loading pruning rewrite rules...
Loading pruning theorems...
Fastest> loadspec symbolTable.tex
Loading specification..
Specification loaded.
Fastest> selop LookUp
Fastest> genalltt
Generating test tree for 'LookUp' operation.
Fastest> addtactic LookUp_DNF_1 SP \in s? \in \dom st
Fastest> genalltt
Fastest> showsch -tcl

\begin{schema}{LookUp\_ DNF\_ 1}\\
 LookUp\_ VIS
\where
 s? \in \dom st
\end{schema}


\begin{schema}{LookUp\_ SP\_ 1}\\
 LookUp\_ DNF\_ 1
\where
 \dom st = \{ \}
\end{schema}


\begin{schema}{LookUp\_ SP\_ 2}\\
 LookUp\_ DNF\_ 1
\where
 \dom st = \{ s? \}
\end{schema}

...
```

Figura 2.3: Comandos para generación de clases de prueba en *Fastest*

En lo que respecta a la generación de lenguaje natural, *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]. La situación propuesta en el trabajo recién mencionado fue una solución ad-hoc, dependiente del dominio de aplicación y de la cantidad de operaciones.

Como mencionamos anteriormente, uno de los objetivos de este trabajo será el de extender la implementación de *Fastest* para permitir al usuario generar descripciones, independientes del dominio de aplicación y de la cantidad de operaciones de las clases de prueba generadas con la herramienta. El sistema de NLG a desarrollar en este trabajo será íntegramente implementado en Java e integrado como una extensión de *Fastest* (trabajando para esto con la versión 1.6 de la herramienta). En el capítulo 8 veremos algunos de los detalles más relevantes de esta implementación.

2.2. 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 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 la sintaxis propuesta por Jackson [Jac95]:

$$\textit{texto informal} \approx \textbf{término\_formal}$$

El símbolo $\approx$ demarca la frontera entre el mundo real (a la izquierda) y el mundo formal o lógico (a la derecha). Del lado derecho 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 (sección 2.1.1), podríamos contar (entre otras) con las siguientes designaciones:

Símbolo a buscar	$\approx$	$s?$
Información asociada a x	$\approx$	$st\ x$

Figura 2.4: Algunas designaciones para *SymbolTable*

Para Jackson, las designaciones sirven en primera instancia cuando se empieza a escribir la especificación para diferenciar un fenómeno en particular y darle un nombre. Luego, le será de utilidad al programador a la hora de leer la especificación. Jackson propone construir un *"puente angosto"* entre la especificación y los elementos del dominio, escribiendo la menor cantidad de designaciones posibles y definiendo otros términos en base a las anteriores.

La función que cumplirán las designaciones en éste trabajo difiere un poco de la propuesta por Jackson. Para nosotros las designaciones resultarán la principal fuente de conocimiento para nuestro sistema de NLG. Y serán fundamentales para que éste pueda generar descripciones independientes del dominio de aplicación.

Veamos, por ejemplo, las siguientes expresiones pertenecientes a dos clases de prueba de dos especificaciones distintas:

1. $\mathrm{dom}\, st = \{s?\}$
2. $\mathrm{dom}\, cajas = \{num?\}$

La primera expresión pertenece a una clase de prueba generada para la operación *LookUp*, la segunda es parte de una clase de prueba generada para una operación perteneciente a la especificación de un sistema bancario. Estas expresiones resultan equivalentes (de hecho, hasta podríamos haber usado los mismos nombres de variables para ambas especificaciones); será gracias a las designaciones que podremos otorgarles una descripción en lenguaje natural (acorde al dominio de aplicación de cada especificación) a cada una de estas expresiones. En particular, para estas, las descripciones podrían ser las siguientes:

1. *"El símbolo a buscar es el único cargado en la tabla de símbolos."*
2. *"El número de caja de ahorro ingresado es el único cargado en el banco."*

Los sistemas de generación de lenguaje natural generalmente utilizan un diccionario 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 sistema de NLG.

Por otro lado, en algunas situaciones, contar con una mayor cantidad de designaciones nos permitirá generar mejores descripciones. Designaciones que podrían resultar redundantes para una persona que lea la especificación podrían, por ejemplo, permitirle a nuestro sistema de NLG generar textos más naturales. En el apéndice B podemos encontrar una pequeña guía sobre qué designar a fin de proveerle la información necesaria a nuestro sistema de NLG para que pueda producir textos más fluidos y naturales.

Por último, cabe mencionar, que es posible que aparezcan parámetros (pertenecientes al término formal) también del lado izquierdo de la designación, como es el caso de la segunda designación presente en la figura 2.4. Llamaremos *designaciones parametrizadas* a este tipo de designaciones, y marcaremos esta diferencia ya que deberemos darle un tratamiento especial a fin de utilizar el texto de estas designaciones en nuestro sistema de NLG. En el capítulo 6.4 desarrollaremos más en detalle esta particularidad.

2.3. Resumen del capítulo

En este capítulo introdujimos conceptos fundamentales para el desarrollo de este trabajo. Presentamos el *test template framework*, definimos las nociones de clase y caso de prueba con las que trabajaremos a lo largo de todo el trabajo. Finalmente profundizamos sobre el rol que cumplirán las designaciones en este trabajo, resultando fundamentales para la generación de descripciones independientes del dominio de aplicación. En el próximo capítulo introduciremos las tareas que deberán ser llevadas a cabo por nuestro sistema de NLG para generar descripciones para las clases de prueba generadas por Fastest.

Capítulo 3

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 castellano o cualquier otra lengua humana a partir de algún tipo de representación no-lingüística de la información a comunicar. Estos sistemas combinan conocimientos tanto del lenguaje en cuestión como del dominio de aplicación para producir automáticamente documentos, reportes, mensajes o cualquier otro tipo de textos.

Dentro de la comunidad de investigación y desarrollo 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 más importantes de esta metodología y en capítulos posteriores desarrollaremos más en profundidad los puntos más relevantes para nuestro trabajo.

3.1. 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 lo mejor posible todas las salidas posibles que se espera que el sistema genere.

En el capítulo 4 profundizaremos más sobre este tema, describiendo y analizando el *corpus de descripciones* utilizado para este trabajo.

3.2. Tareas de la generación de lenguaje natural

La clasificación de Reiter y Dale distingue las siguientes siete tareas que deben ser realizadas a lo largo de todo el proceso de generación de lenguaje natural:

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

Generación de expresiones de referencia: es la tarea de elegir qué 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”, “éste”, etc.

Agregación: se encarga de combinar dos o más elementos informativos con el fin de conseguir un texto más fluido y legible. La agregación decide qué elementos se pueden agrupar para generar oraciones más 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.

3.3. Arquitectura para *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 simultáneo. 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, más comúnmente utilizada para sistemas de NLG, introducida también por Reiter y Dale [RD00]. Esta consiste de tres módulos conectados mediante un *pipeline*. La misma se encuentra en el medio de los dos extremos antes mencionados. En la figura 3.1 podemos observar las tres etapas que componen este *pipeline*; a continuación detallaremos brevemente las tareas a realizar en cada una de estas.

El primer módulo de nuestro sistema será el *document planner*[1], encargado de realizar las tareas de determinación de contenido y estructuración del documento. Veremos en el capítulo 5 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*, éste 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 más refinada del texto final. El *microplanner* será el responsable de transformar los elementos informativos incluidos en el *document plan* en una especificación más concreta de una oración. En el capítulo 6 desarrollaremos más en profundidad el funcionamiento del *microplanner*.

Finalmente el *surface realiser* tomará como entrada la especificación de texto generada 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 antes mencionadas.

[1]Por convención en la comunidad *NLG* nos referiremos a los módulos del pipeline en inglés.

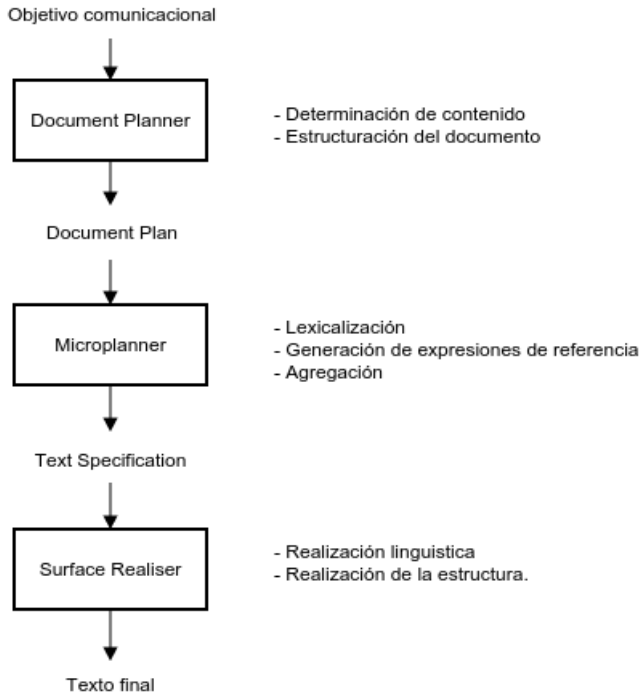

Figura 3.1: Arquitectura típica de un sistema NLG

3.4. Resumen del capítulo

En este capítulo presentamos la metodología propuesta por Reiter y Dale, viendo las tareas básicas que un sistema de generación de lenguaje natural debe llevar a cabo. En el capítulo siguiente, daremos el primer paso necesario para la construcción de este sistema realizando un análisis detallado del corpus de descripciones a fin de obtener los requerimientos necesarios para nuestro trabajo. En los capítulos posteriores profundizaremos sobre el resto de las tareas introducidas en este capítulo: determinación de contenido, *microplanning* y realización de superficie.

Capítulo 4

Análisis de requerimientos

El primer paso en la construcción de cualquier sistema de software, incluyendo los sistemas de generación de lenguaje natural, es realizar un análisis de requerimientos y a partir de estos generar una especificación inicial para el sistema. Para el análisis de requerimientos, seguiremos el enfoque sugerido por Reiter y Dale [RD00] en el cual se propone realizar un corpus de textos de ejemplo y a partir de ellos obtener los requerimientos para nuestro trabajo.

4.1. Corpus de descripciones

Según la definición de la RAE [EdA14], un corpus es un *"conjunto lo más extenso y ordenado posible de datos o textos científicos, literarios, etc., que pueden servir de base a una investigación"*. En particular, el corpus utilizado para la construcción de un sistema de NLG estará formado por un conjunto de ejemplos de datos de entrada junto a la correspondiente salida (texto en lenguaje natural) para cada uno de estos. En nuestro caso, la entrada será un grupo de clases de prueba (en lenguaje Z) junto a las designaciones correspondientes a la especificación testeada, 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 deberá cubrir todo el rango de textos que esperan ser producidos por el sistema de NLG; éste debería cubrir los casos más frecuentes, así como los casos más inusuales que pudieran ocurrir.

Siguiendo la metodología propuesta por Reiter y Dale, para construir el corpus que utilizaremos a lo largo de todo el trabajo deberemos elaborar, en primera instancia, un *corpus inicial* y luego, de ser necesario, deberemos trabajar el mismo a fin de confeccionar un *corpus objetivo* que será con el que finalmente trabajaremos. Para construir un *corpus inicial* deberemos recolectar un conjunto lo suficientemente amplio de ejemplos que debería ser representativo de la variedad de textos que deseamos generar. Luego una persona capacitada (en nuestro caso alguien capaz de leer esquemas Z y con un alto conocimiento del dominio de aplicación) deberá describir en lenguaje natural

los ejemplos antes mencionados. Esta colección de ejemplos y descripciones de los mismos constituirá nuestro *corpus inicial*. Es posible que esta recopilación requiera algunas modificaciones, por ejemplo: podría ocurrir que alguno de los textos resulten técnicamente imposibles de generar y sea necesarios removerlos del corpus, también podrían existir diferentes variantes correspondientes a la misma entrada de texto y deberíamos resolver los posibles conflictos, etc. Llamaremos, finalmente, *corpus objetivo* al resultado de aplicar las modificaciones necesarias al *corpus inicial*. Este *corpus objetivo* es el que utilizaremos para sustentar muchas de las decisiones que deberemos tomar a lo largo de este trabajo. Además, podría ser de utilidad[1] para realizar una evaluación de nuestro sistema una vez desarrollado, comparando los textos generados por nuestra implementación con las descripciones del corpus realizadas por la persona especializada.

En el apéndice A podemos encontrar los textos incluidos en el corpus utilizado para este trabajo. Para elaborar el mismo, recolectamos una colección de clases de prueba generadas con *Fastest* a partir de distintas especificaciones y luego escribimos manualmente cada una de las descripciones para las mismas. En este proceso, intentamos abarcar todo el rango de textos que esperamos que nuestro sistema sea capaz de producir, para esto, tuvimos en cuenta incluir una gran variedad de clases de pruebas de modo que cubran todas las expresiones u operadores de Z contemplados dentro del alcance de este trabajo, considerando también las posibles combinaciones de estas expresiones. Trabajamos también con especificaciones sobre distintos dominios de aplicación a fin de lograr un corpus que nos sea de utilidad para dar con una solución independiente del dominio de aplicación. En total incluimos clases de prueba generadas con *Fastest* para 5 especificaciones distintas; del total se escogieron las más significativas para describir describir en lenguaje natural y se ignoraron aquellas que contenían algún operador no considerado dentro del alcance de este trabajo.

En la figura 4.1 podemos observar uno de los ejemplos incluidos en el corpus de descripciones utilizado para este trabajo. En este caso la clase de prueba *LookUp_SP_1* (generada a partir de la especificación introducida previamente) y las designaciones correspondientes a la especificación en cuestión serán la entrada del sistema de NLG[2], teniendo como salida la descripción en lenguaje natural presente en el ejemplo.

[1] Esto no está contemplado dentro del alcance de este trabajo, pero como veremos en el capítulo 9, podría ser tenido en cuenta en trabajos futuros.

[2] Estrictamente hablando, todas las designaciones de la especificación deberían formar parte de la entrada del sistema de NLG. En este caso, incluimos sólo las designaciones relevantes a fin de simplificar el ejemplo.

- *Clase de prueba para operación LookUp*

LookUp_SP_1

LookUp_VIS

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

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

- *Designaciones SymbolTable*

  - símbolo a buscar $\approx s?$
  - símbolos cargados en la tabla $\approx \text{dom}\, st$

- *Descripción en lenguaje natural para LookUp_SP_1*

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 4.1: Corpus de textos

4.2. Análisis del corpus

En lo que queda de este capítulo nos encargaremos de analizar los ejemplos presentes en el corpus a fin de extraer los requerimientos para nuestro sistema. Este estudio nos ayudará a sustentar muchas las decisiones que tomaremos al realizar el diseño de nuestro sistema de NLG.

Comenzaremos analizando las clases de prueba generadas por *Fastest* y la correlación entre la estructura de las mismas y la estructura de las descripciones en lenguaje natural. Luego estudiaremos la verbalización de las expresiones Z presentes en las clases de prueba, analizaremos el rol de las designaciones y daremos un conjunto de reglas para describir estas expresiones. Por último contemplaremos cuestiones gramaticales que deberemos tener en cuenta para la realización del sistema.

4.3. Estructura de las descripciones

Podemos notar que las clases de prueba generadas por *Fastest* se encuentran formadas siempre por conjunciones de predicados atómicos y que a cada uno de estos le corresponde exactamente una oración en lenguaje natural dentro del texto final. Además, estas oraciones dependen exclusivamente del predicado que describen, es decir, estas no contienen información sobre otros predicados ni hacen referencia a otras frases generadas. En la figura 4.2 podemos evidenciar esta correspondencia.

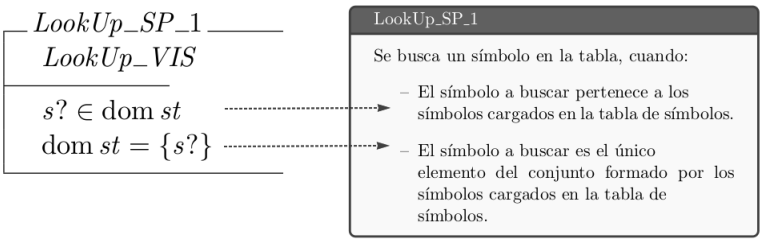

Figura 4.2: Estructura descripciones

Además, podemos observar del corpus (apéndice A) que todas las descripciones de las clases de prueba poseen la misma estructura. En todas se comienza por el nombre de la clase de prueba, seguido por un pequeño detalle de la operación a testear y luego una lista de oraciones encargadas de describir cada uno de los predicados presentes en el cuerpo de la clase de prueba. Conocer la estructura del texto a generar nos será de utilidad para la elaboración del *document plan* (capítulo 5).

4.4. Verbalización de expresiones Z

La tarea de describir un esquema Z correspondiente a una clase de prueba, se puede reducir básicamente a *verbalizar* individualmente cada uno de los predicados atómicos que constituyen el cuerpo del esquema Z. Esta *verbalización* de predicados Z será el desafío principal de este trabajo. Definir con precisión los detalles de ésta será de vital importancia para el desarrollo de las etapas de *microplanning* y realización de superficie de nuestro sistema de NLG (capítulo 6 y 7). En esta sección nos concentraremos especialmente en analizar y especificar la tarea de verbalización.

Podemos notar dos aspectos fundamentales sobre la *verbalización* de expresiones Z. En primer lugar, podemos ver que hay textos que se repiten (con pequeñas variantes, que analizaremos posteriormente) independientemente del dominio de aplicación y que estos surgen a raíz de los operadores de Z presentes en los predicados que se describen. Por otro lado, podemos observar el rol fundamental de las designaciones que posibilitan la introducción de tex-

to dependiente del dominio de aplicación en las descripciones. Consideremos, por ejemplo, las siguientes dos expresiones y sus respectivas descripciones en lenguaje natural, una pertenece al ejemplo de la figura 4.1 y la otra describe un predicado que forma parte de una clase de prueba para la especificación de un sistema bancario:

1. $s? \in \operatorname{dom} st \rightarrow$ *"El símbolo a buscar* ***pertenece a*** *los símbolos cargados en la tabla de símbolos."*
2. $s? \in \operatorname{dom} st \rightarrow$ *"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 $\in$, diferenciándose ambas descripciones en el texto que antecede y sucede al mismo. Por otro lado, estos últimos, bien podrían ser las verbalizaciones de las expresiones que se encuentran a la derecha e izquierda del operador $\in$, respectivamente, por lo que podríamos pensar en una verbalización recursiva sobre la estructura de las expresiones Z. Es posible encontrar estos tipos de patrones en todas las descripciones presentes en el corpus. Esto resulta un buen punto de partida para intentar especificar nuestra tarea de verbalización. En un primer intento, entonces, definiremos la tarea de verbalización en base a los operadores que la componen. Especificaremos esta tarea mediante una función que tomará como entrada una expresión Z y devolverá una descripción en lenguaje natural para la misma. Ésta será recursiva sobre la estructura de las expresiones Z y tendrá tantos casos en su definición como operadores contemplados dentro del alcance de este trabajo (además de las posibles combinaciones de los mismos que puedan resultar de interés y requieran un tratamiento particular). En base al ejemplo anterior, podríamos proponer la siguiente definición para verbalizar el operador $\in$[3,4] :

$$verb'(x \in y) = verb'(x) + \text{"pertenece a"} + verb'(y)^{5}$$

Como mencionamos anteriormente, para verbalizar términos compuestos necesitemos verbalizar recursivamente las partes que componen a los mismos. En el ejemplo anterior necesitaríamos conocer las verbalizaciones de las expresiones x e y para poder obtener una verbalización para $x \in y$.

Retomando el ejemplo de la figura 4.1, nos concentraremos en verbalizar la primera expresión de la clase de prueba:

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

[3]Supondremos que el operador + se encargará de realizar una concatenación de *strings* agregando un espacio entre las mismas.

[4]El nombre de la función se encuentra primado intencionalmente. Más adelante presentaremos la definición final para la función de verbalización que hará uso de `verb'` como función auxiliar.

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

$$verb'(s? \in \mathrm{dom}\, st) \rightarrow verb'(s?) + \text{“pertenece a”} + verb'(\mathrm{dom}\, st)$$

Teniendo que verbalizar $s?$ y $\mathrm{dom}\, st$. En este caso, ambas expresiones se encuentran designadas, lo que sería de gran ayuda para nuestra tarea de verbalización ya que en estas situaciones podremos construir la descripción en base al texto presente en la designación y sin intentar verbalizar la expresión en base al término Z que la compone. En el ejemplo anterior, no deberíamos intentar verbalizar $\mathrm{dom}\, st$ como:

$$verb'(\mathrm{dom}\, st) \rightarrow \text{“el dominio”} + verb(st)$$

de hacerlo, estaríamos perdiendo información valiosa para nuestras descripciones contenida en las designaciones ya que las designaciones son nuestra única forma de introducir texto referente al dominio de aplicación en las descripciones.

En general, para verbalizar una expresión designada, deberemos utilizar el texto presente en las designaciones (en algunos casos, como veremos más adelante, con algunas pequeñas modificaciones, para resolver ciertas cuestiones de concordancia gramatical). Será un requerimiento para nuestra tarea de verbalización, entonces, contemplar en primera instancia si la expresión a describir se encuentra designada antes de intentar describirla en base a los operadores que la componen; de estar designada, deberemos construir una descripción en base a su designación.

Como mencionamos previamente, el nombre de la función anterior (`verb'`) fue primado intencionalmente a fin de utilizar esta función como auxiliar para la definición final de la tarea de verbalización. En la figura 4.3 introducimos una nueva definición para esta tarea donde consideramos la designación de la expresión a verbalizar (si es que se encuentra designada) antes de intentar describir la misma en base a los operadores que la constituyen.

```
verb (exp) = if esta_designada(exp)
             then designacion(exp)
             else verb'(exp)
```

Figura 4.3: Definición verbalización

Como podemos ver en la figura anterior, haremos uso de `verb'` que será la responsable de generar las verbalizaciones para las expresiones de Z en base a un conjunto de reglas dependientes del operador a describir, como la introducida anteriormente para el caso del operador $\in$. Por otro lado, abstraeremos por medio de `esta_designada()` la tarea de verificar si una expresión se en-

cuentra designada y mientras que la función `designacion()` será la encargada de generar una descripción en base a la designación de la expresión a describir.

Tanto para determinar si una expresión se encuentra designada como para generar una descripción para la misma, deberemos considerar el hecho de que ésta podría estar parametrizada. En particular, para la verbalización de una expresión que se encuentra designada por medio de una designación parametrizada, deberíamos considerar tanto el texto presente en la designación como también la verbalización del argumento parametrizado. Por otro lado, de tratarse de una designación no parametrizada podríamos utilizar el texto tal cual se encuentra en la misma. Por ejemplo, teniendo en cuenta las designaciones introducidas en la figura 2.4, obtendríamos la siguiente designación para el término $s?$:

$$designacion(s?) \rightarrow \textit{“símbolo a buscar”}$$

Por otro lado, si se tratase de un término que concuerda con una designación parametrizada, como $st\ s?$, podríamos generar la descripción haciendo uso de su designación y la verbalización del argumento de la siguiente manera:

$$designacion(st\ s?) \rightarrow \textit{“información asociada a”} + verb(s?)$$

4.5. Reglas de verbalización

En esta sección retomaremos al análisis de la función `verb'` presentando una especificación para la misma, basada en un conjunto de reglas (similares a la introducida anteriormente para la verbalización del operador $\in$) para la definición por casos de la misma de la misma. Luego, en lo que queda de este capítulo nos concentraremos en estudiar, como consecuencia de estas reglas, algunos aspectos gramaticales que deberemos contemplar.

Comenzaremos, a continuación, por completar la definición[6] de `verb'`, basándonos en los textos presentes en el corpus, incluyendo los casos para todos los operadores considerados dentro del alcance del trabajo, así como las combinaciones más relevantes de los mismos[7]:

```
verb' ({exp1} = exp2) = verb(exp1) +
                        "es el unico elemento de" +
```

[6] La idea no es que ésta sea una definición final, sino sólo un bosquejo que nos permita analizar los requerimientos que deberán ser contemplados posteriormente, que atenderemos principalmente en las etapas de *microplanning* y realización de superficie.

[7] Para algunos de estos casos necesitamos contemplar el género y número de las verbalizaciones recursivas que lo conforman. En estos casos, nos referiremos al género masculino mediante la letra M, y utilizaremos F para hacer referencia al genero femenino. Por otro lado cuando hagamos referencia al número, utilizaremos S para indicar que el género de un constituyente es singular y P en caso de que sea plural.

```
                          verb(exp2)

verb' (exp1 = {}) = "no hay ningun elemento en" +
                    verb(exp1)

verb' (exp1 ∩ exp2 = {}) = verb(exp1)  +
                           "y" +
                           verb(exp2)  +
                           "no tienen ningun elemento en comun"

verb' (exp1 ∩ {exp2} = {}) = verb(exp2) +
                             "no pertenece a" +
                             verb(exp1)

verb' (exp1 = exp2)
    | (num == S) = verb(exp1) + "es igual a" + verb(exp2)
    | (num == P) = verb(exp1) + "son iguales a" + verb(exp2)
    where num = numero(verb(exp1))

verb' (exp1 ≠ {}) = "existe al menos un elemento en" +
                    verb(exp1)

verb' (exp1 ∩ exp2 ≠ {}) = verb(exp1)  +
                           "y" +
                           verb(exp2) +
                           "tienen al menos un elemento en comun"

verb' (exp1 ≠ exp2)
    | (num == S) = verb(exp1) + "no es igual a" + verb(exp2)
    | (num == P) = verb(exp1) + "no son iguales a" + verb(exp2)
    where num = numero(verb(exp1))

verb' (exp1 < exp2) = verb(exp1) +
                      "es menor a" +
                      verb(exp2)

verb' (exp1 < 0)
    | (gen == M) = verb(exp1) + "es negativo"
    | (gen == F) = verb(exp1) + "es negativa"
    where gen = genero(verb(exp1))

verb' (exp1 ≤ exp2) = verb(exp1) +
                      "es menor o igual a" +
```

```
                        verb(exp2)

verb' (exp1 > exp2) = verb(exp1) +
                        "es mayor a" +
                        verb(exp2)

verb' (exp1 > 0)  =
    | (gen == M) = verb(exp1) + "es positivo"
    | (gen == F) = verb(exp1) + "es positiva"
    where gen = genero(verb(exp1))

verb' (exp1 ≥ exp2) = verb(exp1) +
                        "es mayor o igual a" +
                        verb(exp2)

verb' (exp1 ∈ exp2)
    | (num == S) = verb(exp1) + "pertenece a" + verb(exp2)
    | (num == P) = verb(exp1) + "pertenecen a" + verb(exp2)
    where num = numero(verb(exp1))

verb' (exp1 ∉ exp2)  =
    | (num == S) = verb(exp1) + "no pertenece a" + verb(exp2)
    | (num == P) = verb(exp1) + "no pertenecen a" + verb(exp2)
    where num = numero(verb(exp1))

verb' (exp1 ⊂ exp2)
    | (gen == M && num == S) = verb(exp1) +
                               "esta incluido en" +
                               verb(exp2)
    | (gen == M && num == P) = verb(exp1) +
                               "estan incluidos en" +
                               verb(exp2)
    | (gen == F && num == S) = verb(exp1) +
                               "esta incluida en" +
                               verb(exp2)
    | (gen == F && num == P) = verb(exp1) +
                               "estan incluidas en" +
                               verb(exp2)
    where gen = genero(verb(exp1))
          num = numero(verb(exp1))

verb' ({exp1, ..., expn} ⊂ expm) = verb(exp1) +
                               ", ... , y" +
```

```
                                  verb(exp_n) +
                                  "pertenecen a" +
                                  verb(exp_m)

verb' (exp_1 ⊄ exp_2)  =
    | (gen == M && num == S) = verb(exp_1) +
                                  "no esta incluido en" +
                                  verb(exp_2)
    | (gen == M && num == P) = verb(exp_1) +
                                  "no estan incluidos en" +
                                  verb(exp_2)
    | (gen == F && num == S) = verb(exp_1) +
                                  "no esta incluida en" +
                                  verb(exp_2)
    | (gen == F && num == P) = verb(exp_1) +
                                  "no estan incluidas en" +
                                  verb(exp_2)
    where gen = genero(verb(exp_1))
        num = numero(verb(exp_1))

verb' (exp_1 ⊆ exp_2) = verb(exp_1) +
                    "esta incluido o es igual a" +
                    verb(exp_2)

verb' ({exp_1, ..., exp_n} ⊆ exp_m) = verb(exp_1) +
                                  ", ... , y" +
                                  verb(exp_n) +
                                  "pertenecen a" +
                                  verb(exp_m)

verb' (exp_1 ⊈ exp_2) = "existe al menos un elemento en" +
                     verb(exp_1) +
                     "que no se esta en" +
                     verb(exp_2)

verb' (exp_1 ↦ exp_2) = "el par ordenado formado por:" +
                      verb(exp_1) +
                      "y" +
                      verb(exp_2)

verb' ({}) = "el conjunto vacio"

verb' ({exp_1}) = "el conjunto formado por"  +
```

```
                verb(exp_1)

verb' ({exp_1, ..., exp_n}) = "el conjunto formado por" +
                              verb(exp_1) +
                              ", ... , y" +
                              verb(exp_n)

verb' (exp_1 ∪ exp_2) = "elementos en" +
                         verb(exp_1) +
                         "y en" +
                         verb(exp_2)

verb' (exp_1 ∩ exp_2) = "elementos en" +
                        verb(exp_1) +
                        "que tambien se encuentren en" +
                        verb(exp_2)

verb' (f ~ exp_1) = verb(f) +
                    "aplicada a" +
                    verb(exp_1)

verb' (dom(exp_1)) = "dominio de" +
                     verb(exp_1)

verb' (ran(exp_1)) = "rango de" +
                     verb(exp_1)
```

4.6. Aspectos gramaticales

Como era de esperarse, podemos notar que, a fin de lograr descripciones más 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.

Por otro lado, podemos observar que en algunos casos el texto generado por nuestra verbalización puede variar de acuerdo a aspectos gramaticales, en particular, las distintas variantes de nuestras oraciones dependerán del género y número de los constituyentes de las mismas. En estos casos, nos debemos asegurar que las palabras generadas por nuestro sistema concuerden con rasgos gramaticales de, por ejemplo, otra palabra introducida por medio de una designación. Para esto deberemos considerar algunas reglas de *concordancia gramatical*, como, la concordancia de número entre el verbo y el núcleo del

sujeto para el caso del operador $\in$:

```
verb' (exp1 ∈ exp2)
     | (num == S) = verb(exp1) + "pertenece a" + verb(exp2)
     | (num == P) = verb(exp1) + "pertenecen a" + verb(exp2)
     where num = numero(verb(exp1))
```

En estos casos, deberemos realizar una análisis del sujeto y según corresponda conjugar correctamente el verbo utilizando, para el ejemplo anterior, deberíamos utilizar el verbo "*pertenece*" o "*pertenecen*" según corresponda.

Hay algunos casos de la definición anterior en los que el verbo que forma parte del predicado del texto a generar no requieren ningún tratamiento a fin de que el mismo concuerde en número y forma con el sujeto (esto fue contemplado de antemano y no depende de cuestiones externas como podría ser texto introducido por medio de designaciones). Por ejemplo, los siguientes casos:

```
verb' (exp1 = {})          = "no hay ningun elemento en" +
                              verb(exp1)

verb' (exp1 ∩ exp2 = {}) = verb(exp1)  +
                           "y" +
                           verb(exp2)  +
                           "no tienen ningun elemento en comun"
```

Por otro lado, podemos observar que las oraciones en las que debemos prestar especial atención a cuestiones de concordancia gramatical resultan oraciones bimembres, todas expresadas en tiempo presente, conformadas de la siguiente manera:

sujeto + verbo + objeto

Como por ejemplo, los siguientes casos:

1. `verb'(`*exp*1 $\in$ *exp*2`)` $\rightarrow$ **sujeto** + *"pertenece(n)"* + **objeto**
2. `verb'(`*exp*1 $\subset$ *exp*2`)` $\rightarrow$ **sujeto** + *"está(n) incluido/a(s)"* + **objeto**

Puntualmente observamos que 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. Podemos ver que en el ejemplo anterior la palabra *"incluido"* cumple el rol de atributo del verbo copulativo *"estar"* y deberá concordar en número y forma con el sujeto de la oración.

Otra cuestión que podemos observar a partir de las designaciones documentadas es que, en general para las designaciones no parametrizadas[8], utilizamos frases nominales para escribir las designaciones. Es decir, presentarán la siguiente estructura:

[artículo] + **sustantivo** + [**complemento**]

Como vemos, el artículo puede o no estar presente en las designaciones. Por ejemplo, en ninguna de las designaciones de la figura 4.1 se incluyeron los artículos, pero fue necesario determinar el artículo indicado (de acuerdo al sujeto) y agregarlos en la descripción. Será un requerimiento, entonces, que nuestro sistema identifique los casos en los que sea necesario agregar el artículo apropiado de forma que concuerde en género y número con el sustantivo.

Todas estas observaciones sobre la concordancia gramatical entre los constituyentes de nuestras oraciones serán importantes para el desarrollo del realizador lingüístico que estudiaremos en detalle en el capítulo 7.

4.7. Resumen del capítulo

En este capítulo analizamos los aspectos más destacados que observamos en el corpus de descripciones. Detallamos los requerimientos necesarios para las distintas etapas de nuestro sistema de NLG. Vimos la estructura de las descripciones que utilizaremos en el capítulo 5 donde estudiaremos la tarea de estructuración del documento, estudiamos la verbalización de expresiones y presentamos una especificación para esta tarea que será de vital ayuda para el desarrollo del *microplanner* que estudiaremos en el capítulo 6, finalmente observamos algunos aspectos gramaticales que deberemos contemplar para la realización de superficie en el capítulo 7.

[8]Las designaciones parametrizadas son más complejas de trabajar. Estas estarán formadas generalmente por una oración bimembre que contendrá al parámetro. En el capítulo 6.4 retomaremos sobre este tema y estudiaremos en detalle cómo verbalizarlas.

Capítulo 5

Document Planning

En la arquitectura presentada en el capítulo 3 mencionamos que el *document planner* es el responsable de decidir qué información comunicar (determinación de contenido) y cómo 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 de que la misma se encuentre estructurada de una forma razonablemente coherente. El resultado de esta etapa será una representación del contenido y la estructura del texto final, a la que llamaremos *document plan*.

A continuación detallaremos las tareas que debe realizar el *document planner*, describiremos brevemente la entrada y salida del mismo, definiremos cómo modelar los elementos informativos (pertenecientes a nuestro *document plan*) y finalmente estudiaremos la estructuración del documento.

5.1. Tareas del *document planner*

Como mencionamos anteriormente, el *document planner* será el encargado de llevar a cabo las tareas de *determinación de contenido* y *estructuración de documento*. A continuación describiremos brevemente cada una de estas tres tareas.

Determinación de contenido

La *determinación del contenido* es el nombre que se le da a la tarea de decidir y obtener la información que debemos comunicar en un texto. Este proceso generalmente involucra uno o más procesos de *selección*, *resumen* y *razonamiento* sobre los datos de entrada. A continuación introduciremos estos procesos y posteriormente analizaremos puntualmente cómo deberán ser llevados a cabo en nuestro sistema.

Selección: es el proceso encargado de recopilar un subconjunto de la información de entrada que luego será comunicada al lector. El objetivo de este proceso será el de determinar qué información de la entrada será relevante presentar al lector/usuario final. Veremos más adelante la tarea de selección que deberá ser llevada a cabo por nuestro sistema.

Resumen: es necesario cuando los datos de entrada son demasiado granulados para ser comunicados, o cuando la información relevante consiste de alguna generalización o abstracción de los mismos. En nuestro caso, los datos seleccionados contarán con la información exacta que deseamos comunicar, por lo no será necesario realizar ningún procesamiento de este tipo.

Razonamiento: si bien los dos procesos introducidos anteriormente razonan de cierta forma con los datos de entrada, el objetivo de esta tarea será llevar a cabo un razonamiento más complejo, pretendiendo imitar al razonamiento que podría llevar a cabo un experto en el dominio. En nuestro caso, como veremos más adelante, este razonamiento tendrá que ver con la lógica subyacente de Z.

Para nuestro sistema de NLG será necesario realizar una tarea de *selección* que recopile el conjunto de clases de prueba que debemos describir. Luego veremos que razonando sobre el resultado de la selección será posible obtener mejores descripciones. En particular llevaremos a cabo dos tareas del tipo procesamiento con los datos: *eliminación de tautologías* y *reducción de expresiones*. La primera nos permitirá filtrar expresiones que no añaden información adicional a la descripción final. Por otro lado, la *reducción de expresiones* será la encargada de simplificar algunas expresiones presentes en las clases de prueba. Llevar a cabo estas tareas nos permitirá obtener descripciones más concisas y claras. Tanto las tautologías como las expresiones innecesariamente complejas son resultado del proceso automático de generación de clases de prueba implementado por Fastest. Es importante lidiar con estas lo antes posible ya que hacerlo en etapas posteriores del *pipeline*, además de resultar lingüísticamente más complejo, estaríamos introduciendo procesamiento dependiente de la generación de clases de prueba implementada por Fastest en módulos que deberían encargarse solamente de cuestiones lingüísticas. En la sección 5.4 retomamos la determinación de contenido, describiendo en detalle la solución propuesta en el contexto de este trabajo.

Estructuración del documento

Una vez seleccionada y procesada la información que debemos comunicar, será la tarea de *estructuración de documento* la encargada de agrupar dicha información con el fin de que el texto a generar resulte coherente y posea una estructura que le permita al lector interpretar el contenido con facilidad. Necesitaremos considerar cómo organizar y estructurar la información que

debemos transmitir en el texto final con el fin de producir una descripción razonablemente fácil de leer y comprender. La *estructuración de documento* deberá encargarse de aspectos estructurales a nivel del documento que deseamos producir, donde se contemplarán, por ejemplo, cuestiones como el orden en el que debe ser comunicada la información, cómo estará agrupada la misma, etc. El resultado de la *estructuración de documento* (y del *document planner*), será una estructura intermedia de nuestro *pipeline* con la información antes mencionada a la que llamaremos *document plan*.

En las próximas secciones definiremos detalladamente la entrada y salida del *document planner* estudiando cuáles son los elementos informativos del *document plan* y cómo los modelaremos. Luego veremos las tareas que realizaremos durante la *determinación de contenido* y cómo construiremos nuestro *document plan*.

5.2. Entrada y salida del *document planner*

Como el *document planner* es el primer módulo de nuestro pipeline, la entrada de éste será la misma que la entrada de nuestro sistema. Reiter y Dale [RD00] generalizan la entrada de un sistema de NLG como una cuádrupla formada 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 que los textos generados deberán contener. En nuestro, caso la fuente de conocimiento estará compuesta por la especificación del sistema a testear, las clases de prueba generadas a partir de ésta y las designaciones de la misma.

Objetivo comunicacional: especifica el propósito que debe cumplir el sistema. En general esta compuesto por un "tipo de objetivo" y un parámetro. Para 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 a describir.

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 anteriores con el usuario pueden resultar de utilidad para aumentar la calidad de la generación de lenguaje natural.

Como mencionamos anteriormente, la salida del *document planner* será un *document plan*. En nuestra arquitectura estará estructurado como un árbol,

donde las hojas representarán el contenido y los nodos internos especificarán información estructural, por ejemplo sobre cómo debe agruparse la el contenido a comunicar. Para poder definir esta estructura, deberemos analizar primero cómo representar la información que necesitamos comunicar. En la sección 5.3 estudiaremos cómo representar la información a transmitir y posteriormente, en la sección 5.5, detallaremos la estructura para nuestro *document plan* y veremos en detalle cómo debemos construir el mismo.

5.3. 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 sí mismo la información; la información subyace estos constructores lingüísticos y es "llevada" por ellos. Nos deberemos concentrar entonces en cómo representar este conocimiento y cómo *mapear* estas estructuras a una representación semántica.

En esta sección definiremos los *mensajes* que manipulará nuestro sistema. Llamamos *mensajes* [RD00] a los elementos informativos que conceptualizan la información que queremos comunicar; son paquetes de información que debe estar presente en el texto final. Éstos a su vez estarán compuestos por elementos del dominio de aplicación.

El *corpus de descripciones* (apéndice A) resulta una buena fuente para estudiar el modelado del dominio y los tipos de *mensajes* que necesitamos comunicar. Anteriormente, observamos en el corpus la relación entre las frases pertenecientes a la información a comunicar y las expresiones Z de las clases de prueba generadas por *Fastest*. Observamos que estas clases de prueba están compuestas por una conjunción de predicados atómicos y que cada uno de estos predicados se corresponde con una oración en lenguaje natural dentro de la descripción de la clase de prueba. Podemos ver esta correspondencia en la figura 4.2 (página 34), donde las siguientes expresiones:

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

se encuentran, respectivamente, descritas por las siguientes frases:

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."*

Por otro lado, podemos observar que es posible describir independientemente cada uno de los predicados incluidos en el cuerpo de una clase de prueba. Es por esto que decidimos utilizar un único tipo de *mensaje* para empaquetar cada uno de estos predicados a comunicar: *VerbalizacionExpresion* que

representa, como su nombre lo indica, la verbalización de una expresión Z. Idealmente, tendremos un mensaje *VerbalizacionExpresion* para cada uno de los predicados atómicos pertenecientes a una clase de prueba.

En la figura 5.1 podemos ver cómo quedarían definidos los mensajes para las expresiones mencionadas anteriormente.

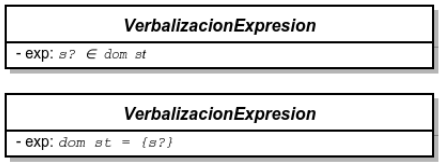

Figura 5.1: Mensajes a comunicar para el ejemplo de la figura 4.2

Vale la pena aclarar que en nuestro caso, los datos con los que trabajamos resultan esquemas Z de las clases de prueba, por lo que ya se encuentran modelados de antemano y no es necesario realizar un nuevo modelo del dominio.

5.4. Determinación del contenido

Como mencionamos anteriormente, en la *determinación de contenido* se suelen llevar a cabo una o más tareas de *selección*, *resumen* y *razonamiento con los datos*. En la figura 5.2 podemos observar el orden en que realizaremos estas tareas y a continuación estudiaremos cada una de ellas detalladamente.

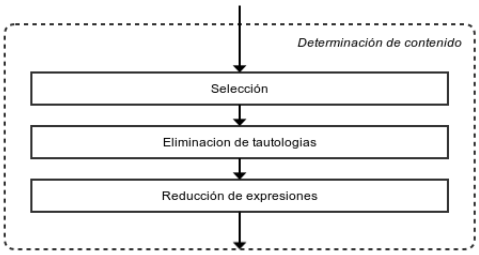

Figura 5.2: Tareas determinación de contenido

Selección

Como vimos en la sección anterior, nuestro sistema de NLG deberá ser capaz de producir descripciones para un subconjunto de clases de prueba del total de las clases generadas por *Fastest* para una especificación. Es decir, el usuario podría solicitarle a nuestro sistema la generación de descripciones para una, un grupo o todas las clases de pruebas generadas y éste debería

generar descripciones únicamente para las clases de prueba indicadas. Es por esto que deberemos realizar una *selección de la información* que tendrá que ser incluida dentro del *document plan* a fin de ser comunicada, posteriormente, en el texto final.

Para el caso de nuestro trabajo esta tarea se resumirá a 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 del sistema. De forma tal que si, por ejemplo, deseamos generar una descripción para la clase de prueba *LookUp_SP_1* (del ejemplo introducido en la sección 2.1.1), la misión de esta tarea será la de identificar y seleccionar la clase *LookUp_SP_1* entre todas las generadas por *Fastest* que formarán parte de los datos de entrada de nuestro sistema de NLG.

Ya seleccionadas las clases de prueba con las que debemos trabajar, veremos cómo podemos mejorar las descripciones de nuestro sistema realizando algunos procesamientos sobre la información seleccionada previo a la construcción del *document plan*. En particular veremos dos tareas que podríamos enmarcar dentro del razonamiento con los datos, la *eliminación de tautologías* y la *reducción de expresiones*. Ambas tareas se realizarán sobre la estructura de los predicados Z, implementando reglas de reescritura para cada uno de los casos. El capítulo 8 presentaremos como éstas se encuentran implementadas y como podremos hacer para agregar nuevas reglas de reescritura a nuestro sistema de NLG[1].

Eliminación de tautologías

Como mencionamos anteriormente, todas las clases de prueba incluidas en el corpus fueron generadas utilizando *Fastest 1.6*. Hemos observado que en ciertos casos, esta herramienta genera clases de prueba como la que podemos ver a continuación:

[1]Las dos tareas de razonamiento trabajan sobre los casos más comúnmente observados dentro del corpus recolectado. Sin embargo, como veremos en el capítulo 9, creemos que será posible mejorar la calidad de las descripciones producidas profundizando sobre la tarea de reducción de expresiones y agregando nuevas nuevas tareas de razonamiento a nuestro sistema.

Update_SP_2

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

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

Figura 5.3: Clase de prueba para operación Update_SP_2

Podemos observar en este caso, que la siguiente expresión del ejemplo anterior:

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

no aporta información relevante para el usuario, de hecho esta expresión no agrega ninguna restricción para el caso de prueba ya que será siempre verdadera y si intentáramos describir este predicado, terminaríamos con un texto parecido al siguiente:

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

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

Esto sugiere que obtendremos descripciones más claras si filtramos este tipo de expresiones. En particular, lo deberíamos hacer lo antes posible en el *pipeline* de nuestro sistema y la tarea de determinación de contenido resulta la apropiada para realizar este tipo de procesamiento. De esta forma evitaremos que etapas posteriores, como la de *microplanning* o *realización de superficie* deban ocuparse de la generación de frases que no aportarían más que confusión al texto final.

En la versión de *Fastest* utilizada en este trabajo solo observamos la aparición de tautologías similares a la del ejemplo anterior con predicados con la siguiente estructura:

$$\{a, b, ..., n\} \neq \{\}$$

por lo tanto, la implementación de nuestro sistema solo necesitará considerar el caso antes mencionado.

En la figura 5.4 podemos ver ilustrado el comportamiento esperado de la tarea de *eliminación de tautologías* para el ejemplo utilizado anteriormente.

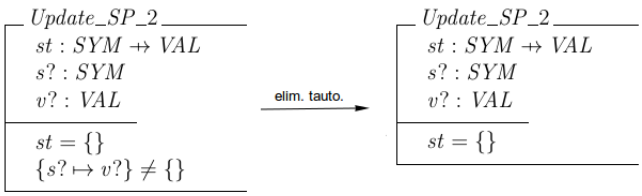

Figura 5.4: Eliminación de tautología para Update_SP_2

Será tarea de nuestro *document planner*, entonces, filtrar tautologías presentes en las clases de prueba para asegurarnos de este modo que no sean incluidas dentro del *document plan*.

Reducción de expresiones

Podemos observar en el corpus que hay algunas expresiones que podríamos simplificar y de esta forma lograr descripciones más claras y concisas. Veamos por ejemplo la figura 5.5.

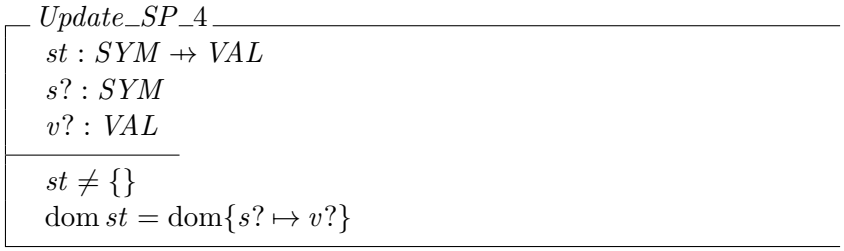

Figura 5.5: Clase de prueba para operación Update

En particular observemos la expresión:

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

Si nos adelantamos un poco y tratamos de verbalizar esta expresión de acuerdo a las reglas presentadas en el capítulo 4.5 podríamos generar una frase como la siguiente:

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

que no parece ser la verbalización más adecuada para la expresión dada. Por otro lado, veamos que es posible reducir la expresión anterior a la siguiente expresión equivalente:

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

esta última expresión resultará más fácil de verbalizar, al menos según las reglas introducidas, nuestro sistema podría generar una descripción similar a la siguiente:

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

El caso presentado anteriormente fue un caso muy recurrente que notamos en las clases generadas por *Fastest*. Este resulta de la aplicación de la táctica de partición estándar sobre el operador $\oplus$, lo cual es bastante habitual. Es por esto que creemos importante trabajar este tipo de expresiones antes de incluirlas dentro del *document plan*. Para esto nuestro *document planner* deberá realizar el siguiente remplazo siempre que sea posible:

$$\text{dom}\{x \mapsto y\} \rightarrow \{x\}$$

En la figura 5.6 podemos ver ilustrado el comportamiento esperado de la tarea de *reducción de expresiones* para el ejemplo utilizado anteriormente.

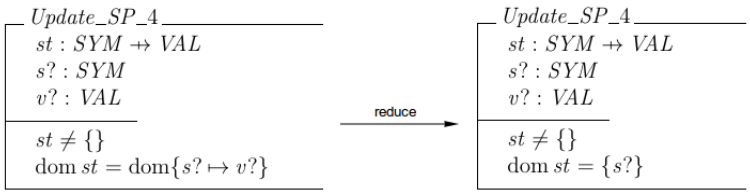

Figura 5.6: Reducción de expresiones para Update_SP_4

Será tarea de nuestro *document planner*, entonces, trabajar también este tipo de expresiones antes de construir los mensajes a incluir dentro del *document plan*.

El caso anterior resulta el más comúnmente observado dentro de las clases de prueba generadas por Fastest y fue el único contemplado dentro del alcance de este trabajo. Sin embargo, veremos en el capítulo 8, cómo fácilmente podremos introducir a nuestra implementación nuevas reglas, similares a la presentada anteriormente.

5.5. Estructuración del documento

Como mencionamos anteriormente, el texto generado no podrá ser una colección de frases y palabras al azar. Deberá tener coherencia y poseer una estructura que le permita al lector interpretar con facilidad el contenido del

mismo. Para esto, necesitaremos considerar cómo organizaremos y estructuraremos la información que debemos comunicar a fin de producir un texto razonablemente fácil de leer y comprender.

Esta tarea se concentrará en construir una estructura que contenga la información seleccionada y procesada en la etapa de *determinación de contenido*, estableciendo el agrupamiento y ordenamiento de la misma. Esta estructura deberá caracterizar la disposición de los elementos pertenecientes a los textos recopilados en el corpus. De éste, podemos observar que los documentos a generar poseen una estructura bastante simple y rígida a la vez: están formados por una secuencia de descripciones para las clases de prueba seleccionadas en la etapa de *determinación de contenido*. A su vez, cada una de estas descripciones agrupa los *mensajes* que modelan la verbalización de las expresiones pertenecientes a cada clase, ordenados de la misma forma en la que aparecen en el esquema de la clase de prueba en cuestión.

Para modelar nuestro *document plan*, utilizaremos un elemento al que llamaremos *DocumentoDP*, este elemento modelará el documento completo (será la raíz de nuestro *document plan*), éste elemento contendrá el título para el documento a generar y un conjunto de elementos que modelarán las descripciones de las distintas clases de prueba. Llamaremos *DescripcionClasePrueba* a estos últimos, utilizados para modelar las descripciones de las distintas clases de prueba. Estos estarán formados por información general de la clase de prueba a describir (como el nombre de la misma y una pequeña descripción de la operación a testear[2]) y un conjunto de los *mensajes* introducidos anteriormente, *VerbalizacionExpresion*, que modelaran las verbalizaciones de las expresiones Z contenidas dentro de las clases de prueba. En la figura 5.7 podemos observar una representación abstracta de la estructura que tendrá nuestro *document plan*.

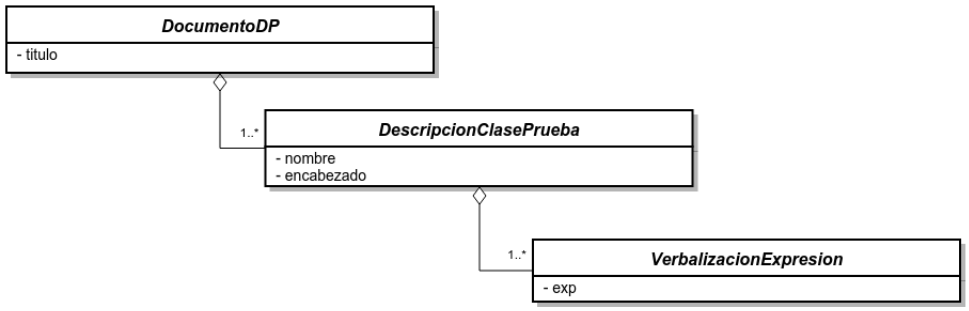

Figura 5.7: *Document plan*

Podemos ver en la figura 5.8 un ejemplo del document plan para la des-

[2]Esta descripción o encabezado en las *DescripcionClasePrueba* se construirá en base al texto utilizado para designar la operación para la cual fue derivada la clase de prueba en cuestión.

cripción de la clase de prueba *LookUp_SP_1* utilizada en el capítulo anterior (página 34).

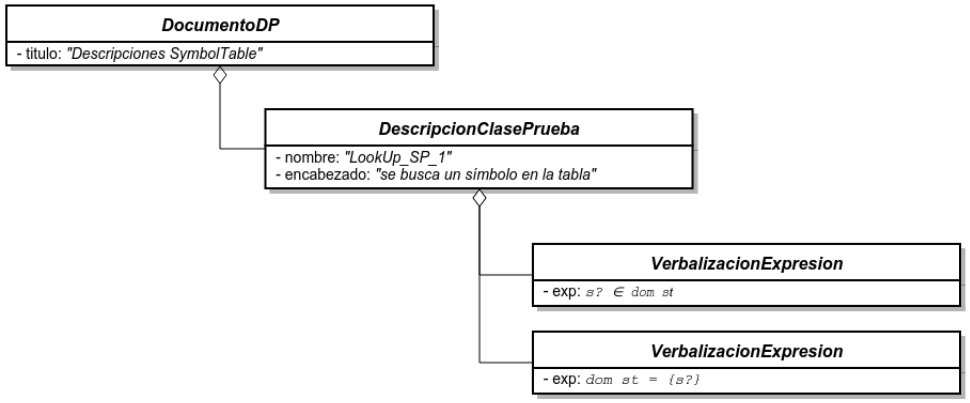

Figura 5.8: *Document plan* correspondiente al texto de la figura 4.2

5.6. Resumen del capítulo

En este capítulo analizamos las tareas que debe realizar nuestro *document planner*. Como vimos, la misión del mismo es construir un *document plan* que contenga la información requerida por el usuario, filtrada, procesada y organizada. Tomamos decisiones generales sobre la estructura del documento, dejando para etapas posteriores el trabajo más detallado, por ejemplo, a nivel de las oraciones. Será tarea del *microplanner*, como veremos el próximo capítulo, procesar los *mensajes* construidos en esta etapa y generar a partir de estos una especificación de frase para cada expresión de Z que debemos verbalizar, así como también transformar los elementos que contienen información estructural en especificaciones más concretas del texto a generar (utilizando elementos que modelarán párrafos, secciones, lista de ítems, etc.).

Capítulo 6

Microplanning

La etapa de *microplanning* será la encargada de, a partir del *document plan* producido por la etapa anterior, producir una especificación más detallada del el texto a generar.

En éste capítulo presentaremos las tres tareas que, según Reiter y Dale [RD00], deberían llevarse a cabo en esta etapa: lexicalización, agregación y generación de expresiones de referencia. Luego definiremos en detalle la entrada y salida del *microplanner*. Finalmente profundizaremos particularmente sobre la tarea de lexicalización que debemos llevar a cabo para este trabajo.

A lo largo de este capítulo continuaremos con el ejemplo utilizado en la etapa anterior, ilustrando cómo a partir del *document plan* presentado en la la figura 5.8 construiremos una especificación más detallada del documento a generar.

6.1. Tareas del *Microplanner*

Como mencionamos previamente, el *microplanner* será el encargado de transformar el *document plan* generado en la etapa anterior en una especificación más refinada del texto a generar. Cabe aclarar que el resultado de esta etapa no será todavía el texto final ya 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 3, dentro de las tareas generalmente realizadas por el *microplanner* podemos destacar las siguientes:

Lexicalización: esta tarea se encarga de elegir qué palabras particulares y qué 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 6.3

Agregación: la función de esta tarea es la de combinar los elementos informativos del *document plan* con el fin de conseguir un texto más fluido y legible.

La agregación decide qué elementos se pueden agrupar para generar oraciones generalmente más complejas sin modificar el significado de las mismas. Por ejemplo, consideremos las siguientes dos descripciones posibles para la clase de prueba *LookUp_SP_1* (figura 4.1):

1. *El símbolo a buscar pertenece a los símbolos cargados en la tabla. El símbolo a buscar es el único elemento de los símbolos cargados en la tabla.*
2. *El símbolo a buscar pertenece a los símbolos cargados en la tabla y éste es el único elemento de los símbolos cargados en la tabla.*

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

Generación de expresiones de referencia: esta tarea se encarga de determinar qué frases deben ser usadas para identificar las diferentes menciones al mismo elemento en un texto a fin de aportar fluidez a éste. Más específicamente, en los casos que se hace referencia a una entidad que ya ha aparecido en el texto se puede remplazar la misma por otra frase que la referencie. La elección de qué expresión utilizar para referirse a la entidad dependerá del contexto y deberá hacerse sin introducir ambigüedad para el lector. Por ejemplo, en la segunda alternativa presentada anteriormente para describir la clase de prueba *LookUp_SP_1*:

El símbolo a buscar *pertenece a los símbolos cargados en la tabla y* ***éste*** *es el único elemento de los símbolos cargados en la tabla.*

Reemplazamos la segunda ocurrencia de "el símbolo a buscar" por el pronombre "éste".

Al igual que la tarea de agregación, la generación de expresiones de referencia excede el alcance de este trabajo y nuestro *microplanner* no realizará tareas este tipo. Sin embargo, en el capítulo 9 propondremos, como un posible trabajo a futuro, la inclusión de las tareas agregación y generación de expresiones de referencia a nuestro sistema.

6.2. Entrada y salida del *microplanner*

La entrada del *microplanner* será un *document plan* producido por la etapa anterior. Observemos por ejemplo el *document plan* presentado en la figura 5.8

del capítulo anterior, utilizado para modelar la descripción de la clase de prueba *LookUp_SP_1*. Esta abstracción no especifica las frases que nuestro sistema debe generar, ni si deben estar enumeradas en una lista de ítems o agrupadas en secciones, por ejemplo. Necesitaremos una especificación más concreta, un modelo más detallado del documento y de las frases a generar. Será entonces responsabilidad del *microplanner* construir a partir del *document plan* una especificación más concreta del texto a generar.

Llamaremos especificación del texto a la especificación resultado de esta etapa. Ésta se encargará de modelar los distintos elementos que compondrán el texto final (como párrafos, lista de ítems, etc.). Esta especificación estará compuesta en base a especificaciones de frase encargadas de modelar las distintas oraciones que serán incluidas en el texto final (veremos que cada una de estas se construirán a partir de los mensajes contenidos en el *document plan*). Será luego tarea de la siguiente etapa convertir los nodos internos de la especificación del texto en anotaciones especificas para el sistema de presentación[1] (realización de estructura) y transformar las especificaciones de frase en oraciones o frases sintáctica, morfológica y ortográficamente correctas (realización lingüística).

En lo que queda de esta sección estudiaremos cómo se encuentra constituida nuestra especificación del texto, describiendo también cómo están formadas nuestras especificaciones de frase.

6.2.1. Especificación del texto

La especificación de texto para nuestro sistema, deberá caracterizar la estructura del documento final que nuestro sistema debe producir. Es por esto que modelaremos los mismos utilizando un árbol, donde los hojas especificarán las frases u oraciones a generar (las especificaciones de frase), y los nodos internos establecerán cómo estas frases tendrán que ser agrupadas en elementos del documento (como párrafos, secciones, lista de ítems, etc).

La estructura de los documentos que debemos generar en este trabajo resulta relativamente simple. Como vimos en el capítulo 4, los documentos de descripciones poseen un título y luego se detallan una por una las descripciones de las distintas clases de prueba, donde para cada una de éstas aparece el nombre de la clase de prueba, junto a una pequeña descripción de la operación a testear y luego una lista de ítems que describirán cada una de las restricciones pertenecientes a la clase de prueba que se describe. Es por esto que para este trabajo utilizaremos sólo dos elementos para modelar la estructura interna del documento *TSDocumento* y *TSListaItems*:

[1] Llamamos sistemas de presentación a los sistemas encargados de post-procesar las anotaciones antes mencionadas y presentarle al usuario el documento de manera apropiada. Por ejemplo, LaTeX, Microsoft Word o web browsers como Firefox o Chrome son algunos de los posibles sistemas de presentación.

TSDocumento: modela el documento final, por lo tanto solo tendremos un elemento de este tipo en nuestra especificación del texto y éste será la raíz del documento. Éste elemento contendrá información general sobre el documento, como el título y una especificación para cada descripción de clase de prueba, modeladas mediante *TSListaItems*.

TSListaItems: modela el texto que describirá a una clase de prueba. Este elemento contiene una especificación de frase que modelará el texto correspondiente al título y al detalle de la descripción. Además contendrá una lista de especificaciones de frase que especificarán las frases para cada una de las verbalizaciones de las expresiones contenidas en la clase de prueba en cuestión. Habrá una *TSListaItems* por cada clase de prueba a describir.

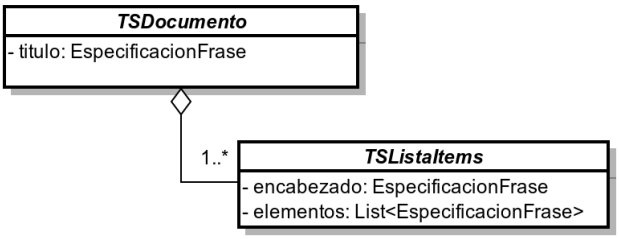

Figura 6.1: Especificación de texto

En la figura anterior podemos observar la estructura abstracta que tendrán nuestras especificaciones del texto y por ejemplo, sin meternos en detalle todavía sobre la estructura de las especificaciones de frase, en la figura 6.2 podemos ver una especificación de frase para el ejemplo introducido anteriormente (página 33).

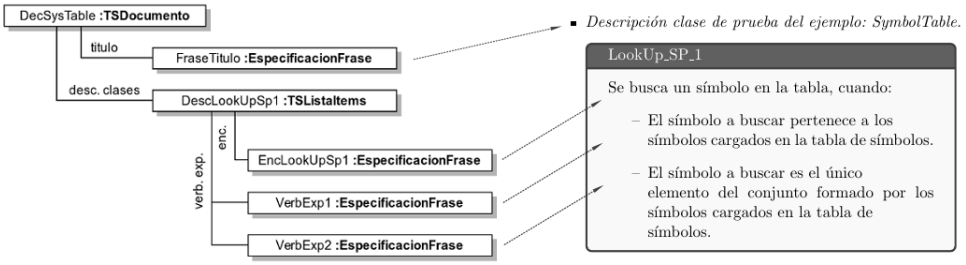

Figura 6.2: Ejemplo especificación del texto

Veremos en el próximo capítulo cómo, en la etapa de realización estructural, transformaremos estos elementos en anotaciones para el sistema de presentación.

6.2.2. Especificación de frase

En la literatura sobre NLG podemos encontrar muchas alternativas en lo que respecta a la especificación de frases (*skeletal propositions*, *meaning specifications* y *lexicalised case frames* entre otras [RD00]). Todas éstas varían en el nivel de abstracción que poseen. Las representaciones más abstractas le darán más flexibilidad a las etapas de *document planning* y *microplanning*, pero al mismo tiempo nos obligarán a tener un realizador de superficie más sofisticado. Por otro lado, las especificaciones menos abstractas, requieren que el *document planner* y el *microplanner* realicen un mayor trabajo, pero también tendrán más 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 de la lengua (castellano en nuestro caso) que queremos generar y no en términos específicos de Z. Es por esto que decidimos especificar las oraciones a generar mediante árboles sintácticos, donde los constituyentes de éstos son los sintagmas de la oración que deseamos generar. Esto le dará la posibilidad al realizador lingüístico de poder identificar la función de cada uno de los constituyentes de la oración. Por ejemplo, como detallamos en los requerimientos de la sección 4.6, el *realizador de superficie* necesitará identificar el núcleo de un sintagma nominal (núcleo del sujeto) para poder producir una oración en la que haya concordancia de número y persona entre el verbo y el sujeto. Como consecuencia del ejemplo anterior, nuestro realizador lingüístico deberá proveerle al realizador de superficie una especificación que permita identificar el sujeto, predicado y verbo de una oración. Creemos que con los elementos presentes en la figura 6.3 podremos modelar la frases incluidas en el corpus.

No pretendemos modelar toda la lengua castellana con estos elementos sino solo un subconjunto que nos provea las herramientas necesarias para permitirle al *realizador de superficie* generar las frases definidas en el capítulo 4.2, ya que el desarrollo de un realizador lingüístico que considere todas las construcciones sintácticas de nuestra lengua escapa el alcance de este trabajo. Es por esto que sólo modelamos los sintagmas nominales (FraseNominal) y verbales (FraseVerbal) y nos vemos 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 sobre la realización de los mismos en el capítulo 7.4.

- ***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.

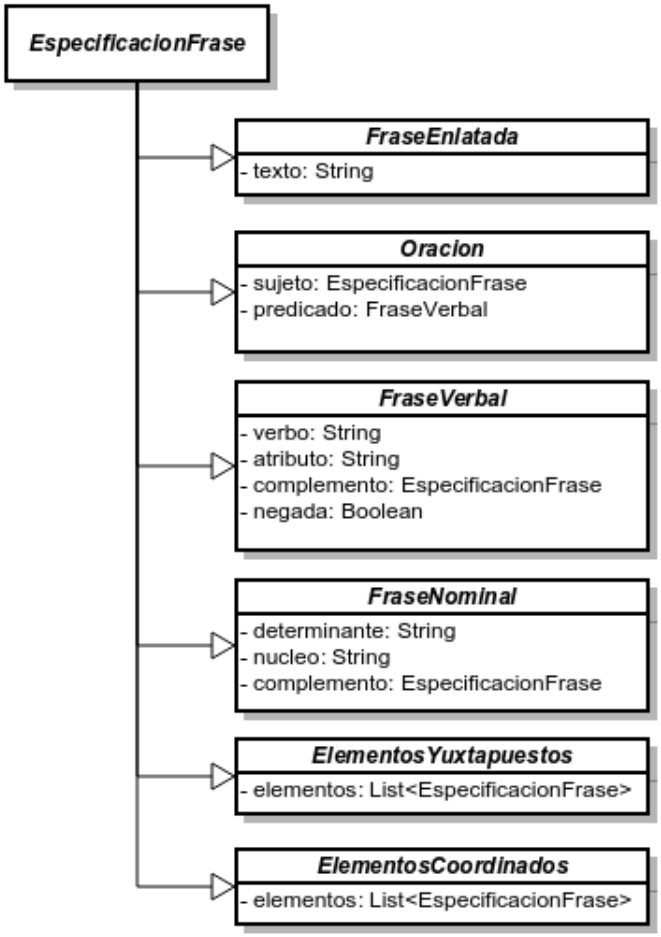

Figura 6.3: Especificación de frase

- ***FraseVerbal***: Representa un sintagma verbal que corresponderá al predicado de una *Oración*.
- ***FraseNominal***: Modela un sintagma nominal. Generalmente conformará el sujeto en una *Oración*.
- ***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* 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.

En la siguiente sección veremos cómo la tarea de lexicalización construirá es-

tas especificaciones de frase en base a la información contenida en los mensajes del *document plan*.

6.3. Lexicalización

Como mencionamos anteriormente, el proceso de lexicalización será el encargado de elegir qué 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 contemplando todos los casos definidos en el capítulo 4.5, es decir, nuestro proceso de lexicalización tendrá que comportarse de forma similar a la función `verb` que estudiamos durante el análisis de requerimientos. Tanto las palabras, como los sintagmas a utilizar se desprenderán de las frases presentes en la definición de `verb`.

Como analizamos en el capítulo 4, nuestro sistema deberá producir una oración en lenguaje natural para cada predicado incluido dentro del cuerpo de cada clase de prueba, a su vez, la verbalización para cada una de estas expresiones se encuentra caracterizada por un mensaje dentro del *document plan*. Es por esto que el módulo encargado de esta tarea deberá ser capaz de generar una *especificación de frase* a partir de la expresión Z contenida en cada uno de estos mensajes. De acuerdo a los requerimientos introducidos en el capítulo 4, nuestro lexicalizador primero deberá verificar si la expresión en cuestión se encuentra designada, en este caso, tendrá que construir una especificación de frase en base a su designación. De lo contrario deberá intentar construirla recursivamente contemplando los casos para los distintos operadores y las posibles combinaciones. En el algoritmo 1 podemos ver un bosquejo del comportamiento esperado para esta tarea, de acuerdo al análisis realizado en la sección 4.5, trabajando esta vez con las especificaciones de frase definidas en la sección anterior. Incluimos sólo un bosquejo ya que ilustrar el comportamiento completo de esta tarea resulta extenso debido a la construcción y composición de elementos que debemos realizar para cada caso; en particular incluimos un caso para la lexicalización del operador $\in$ que utilizaremos posteriormente para desarrollar el ejemplo que presentaremos en la figura 6.4.

Algoritmo 1 Bosquejo de LEXICALIZACION para el operador ∈

```
1: function LEXICALIZACION(exp)
2:     if ESTA_DESIGNADA(exp) then
3:         ret ← DESIGNACION(exp)
4:     else
5:         ret ← LEXICALIZACION'(exp)
6:     end if
7:     return ret
8: end function

9: function LEXICALIZACION'(x ∈ y)
10:    oracion.sujeto ← LEXICALIZACION(x)
11:    fraseVerbal.verbo ← "pertenece"
12:    fraseEnlatada.texto ← "a"
13:    elemYuxtapuesto.elementos ← {fraseEnlatada, LEXICALIZACION(y)}
14:    fraseVerbal.complemento ← elemYuxtapuesto
15:    oracion.predicado ← fraseVerbal
16:    return oracion
17: end function
```

La función DESIGNACION deberá ser capaz de construir una especificación de frase a partir de una expresión designada. Analizaremos este caso con mayor profundidad en la siguiente sección. Por otro lado, notemos que en el caso que la expresión a lexicalizar no se encuentre designada, se deberá analizar recursivamente la expresión para generar el texto adecuado.

A continuación, retomaremos el ejemplo de la figura 6.1 y veremos cómo deberá nuestro sistema generar la especificación de frase para uno de los mensajes incluido en el *document plan*. En la figura 6.4 podemos observar este mensaje y el resultado de la lexicalización del mismo. Veremos a continuación los pasos que deberá realizar nuestro sistema durante la tarea de lexicalización para lograr el resultado ilustrado en la imagen.

El objetivo del lexicalizador será construir una especificación de frase escogiendo adecuadamente las palabras y constructores sintácticos para siguiente la expresión:

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

Asumimos que contaremos con las siguientes designaciones:

$s?$	≈ el símbolo a buscar
$dom\ st$	≈ símbolos cargados en la tabla de símbolos

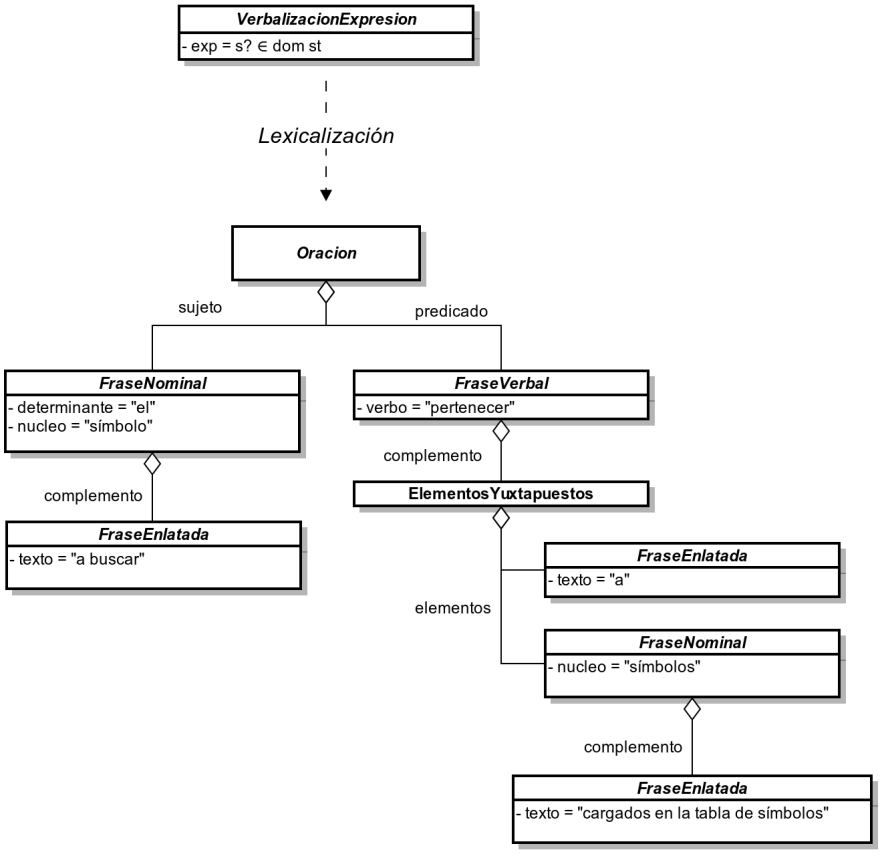

Figura 6.4: Especificación de frase para $s? \in \operatorname{dom} st$

En primer instancia, al no encontrarse designada $s? \in \operatorname{dom} st$ nuestro lexicalizador intentará construir la especificación en base a los operadores que componen la expresión a describir. En el algoritmo 1 incluimos el caso para lexicalizar el predicado $x \in y$ que deberemos utilizar para lexicalizar $s? \in \operatorname{dom} st$.

Será necesario lexicalizar recursivamente las expresiones: $s?$ y $\operatorname{dom} st$. El resultado de estas lexicalizaciones será usado para construir el sujeto y parte del predicado de la oración a especificar. Estas expresiones se encuentran ambas designadas y por lo tanto su especificación de frase se construirá a partir del texto incluido en sus designaciones. Veremos luego en la siguiente sección la tarea de la función DESIGNACION utilizada en el algoritmo anterior encargada de producir una especificación de frase a partir de una expresión designada.

Como ya mencionamos, utilizaremos el resultado de la lexicalización de $s?$ para formar el sujeto de la oración que deseamos especificar y deberemos luego construir la *Frase Verbal* que cumplirá el rol de predicado. Para construir

esta última usaremos la especificación de frase que modelará la verbalización dom st, también mencionada anteriormente.

Finalmente escogeremos las palabras indicadas a fin de que nuestro sistema pueda dar con una descripción similar a la presentada en el capítulo 4.5 del análisis de requerimientos. Notemos que para esto utilizamos el verbo "pertenecer", en infinitivo, siendo luego tarea del realizador lingüístico la de conjugar el mismo de acuerdo a las reglas gramaticales introducidas en la sección 4.6. 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 deberá procesar los elementos contenidos en cada *ElementoYuxtapuestos* generando un texto resultado de la concatenación de la realización los mismos.

A continuación, en la siguiente sección estudiaremos finalmente, la lexicalización de expresiones designadas.

6.4. Lexicalización de expresiones designadas

Como vimos en la sección anterior, nuestra tarea de lexicalización deberá hacer uso de las designaciones presentes en la especificación para la construir una especificación de frase. Para esto, cuando una expresión se encuentre designada, nuestro sistema tendrá que procesar la designación, construyendo una especificación de frase que la caracterice. Esto será necesario ya que, como mencionamos previamente, en la etapa de *realización de superficie* nuestro sistema necesitará conocer los distintos constituyentes sintácticos de las oraciones que les provee la especificación de frase y en algunos casos también deberá modificar levemente los textos presentes en las designaciones (por ejemplo, como mencionamos en la sección 4.6, puede ser necesario agregarle el artículo correspondiente a la frase utilizada en la designación).

Para este trabajo, estudiaremos por separado las designaciones parametrizadas y las no parametrizadas.

Comencemos por analizar la lexicalización de una expresión que se encuentra designada por medio de una designación no parametrizada. Por ejemplo, supongamos que queremos construir una especificación de frase para la expresión $s?$ del ejemplo utilizado en la sección anterior. Recordemos que la designación para la misma es:

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

La oración utilizada en la designación anterior, como en los casos observados en el corpus (para designaciones no parametrizadas) resulta un *sintagma nominal*. En este caso "símbolo" es el núcleo, "el" cumple la función de determinante y "a buscar" es el complemento. Será posible entonces, para nuestra tarea de lexicalización, modelar estas frases utilizando una *FraseNominal*.

Para que nuestro sistema sea capaz de esto deberá analizar sintácticamente las designaciones, *parseando* las mismas con la ayuda de un analizador morfológico que nos permitirá obtener la función sintáctica de cada constituyente de la frase. Además de esto, para simplificar la tarea de *parseo* de nuestro sistema, requeriremos que el usuario escriba las designaciones mediante un sintagma nominal. Es decir, respetando la siguiente estructura:

Sintagma Nominal = [**Determinante**] + **Núcleo** + [**Complemento**]

En el capítulo 8 veremos más en detalle cómo utilizamos un analizador morfológico para ayudarnos con la especificación de las frases incluidas en las designaciones.

Por otro lado, las frases incluidas en las designaciones parametrizadas no poseen la misma estructura. La tarea de modelar minuciosamente estos textos resulta más compleja que para el caso anterior. Por ejemplo, podríamos tener uno o más parámetros presentes dentro del texto, para los cuales deberíamos identificar el rol que cumple cada uno de los anteriores dentro de la oración. Además, nuestro realizador lingüístico sólo soportará oraciones de la forma SVO (sujeto, verbo, objeto) lo cual podría no respetarse en una designación introducida por el usuario. Es por esto que nuestro sistema proveerá solo soporte parcial para las designaciones parametrizadas, aceptando sólo designaciones con un único parámetro. Para describir una expresión parametrizada requeriremos a su vez que el argumento de la misma también se encuentre designado. De esta forma podríamos describir una designación parametrizada de la misma forma que vimos en el capítulo 4 reemplazando el parámetro presente en el texto de la designación por el texto incluido en la designación del argumento.

Veamos por ejemplo las siguientes designaciones para una especificación que modela un pequeño sistema de monitoreo de sensores[2] incluido en el corpus (apéndice A):

$x \in \text{dom}\, smax$	$\approx$ x es un identificador válido
$s?$	$\approx$ el identificador del sensor leído

Donde para describir la expresión $s? \in \text{dom}\, smax$ bastará con reemplazar el parámetro dentro del texto de la designación parametrizada con el texto incluido en la designación de $s?$ como vemos en la figura 6.5.

[2]Cambiamos el dominio debido a que el ejemplo de la tabla de símbolos no presenta un caso adecuado para ilustrar este problema.

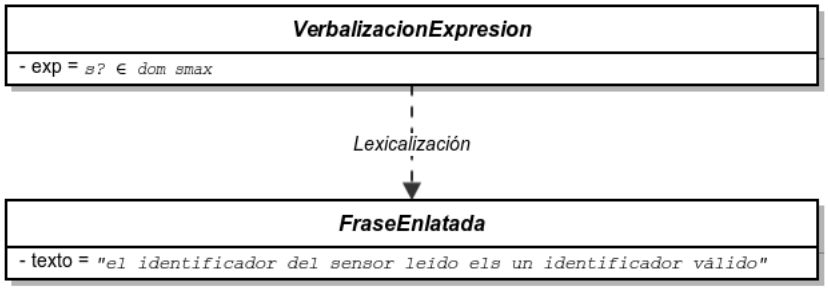

Figura 6.5: Lexicalización $s? \in \text{dom}\, smax$

Como podemos observar en el ejemplo anterior, al desconocer la estructura del texto presente en las designaciones parametrizadas, deberemos modelar el texto producido por la composición de ambas designaciones utilizando una *FraseEnlatada*. Luego, el texto contenido en estos elementos será realizado sin ningún procesamiento previo, por lo que será responsabilidad del especificador (encargado de escribir las descripciones) que el mismo sea sintáctica y ortográficamente correcto.

6.5. Resumen del capítulo

En este capítulo vimos las tareas necesarias para, partiendo de la salida producida por el *document planner*, constituir una especificación más refinada del texto final. En el próximo capítulo veremos finalmente las tareas que deberán llevarse a cabo para transformar esta especificación del texto en el documento final que contendrá todas las descripciones requeridas por el usuario.

Capítulo 7

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 nos concentraremos en estudiar las tareas que deben ser realizadas a fin de transformar esta especificación en texto de superficie, formado por frases, signos de puntuación y algunas etiquetas de *mark-up* necesarias para estructurar el texto.

En las próximas secciones introduciremos las tareas que deberemos llevar a cabo durante esta etapa, repasaremos brevemente la entrada y salida de la misma y finalmente estudiaremos en detalle las tareas de *realización estructural* y *realización lingüística* realizadas en este trabajo.

7.1. Tareas del realizador de superficie

Como mencionamos en el capítulo 3, el realizador de superficie será el encargado de transformar la especificación del texto producida en la etapa de *microplanning* en el documento final. Deberá transformar esta representación abstracta del documento en frases, signos de puntuación y algunas etiquetas de *mark-up* necesarias[1]. Estas últimas dependerán del sistema de presentación utilizado (un archivo de texto, rtf, word, latex, pdf, etc.).

En el capítulo anterior vimos que la especificación del documento es un árbol, cuyas hojas son especificaciones de frases y sus nodos internos representan cómo las anteriores deberán estar agrupadas en el documento final (en general estos nodos representan secciones, párrafos, listas de ítems, etc). Para transformar esta especificación en texto de superficie, nuestro realizador deberá llevar a cabo dos tareas: *realización estructural* y *realización lingüística*. La primera se encargará de *mapear* los nodos internos antes mencionados en anotaciones para el sistema de presentación a utilizar. Por otro lado, la rea-

[1] Por ejemplo, para delimitar un nuevo párrafo en LaTeXdeberemos introducir una línea en blanco antes del comienzo del mismo, para hacerlo en un documento HTML tendremos que delimitar el mismo utilizando las etiquetas *¡P¿* y *¡/P¿*.

lización lingüística deberá convertir las especificaciones de frase en oraciones sintáctica, morfológica y ortográficamente correctas.

En la sección 7.3 veremos en detalle la tarea de *realización estructural* que llevaremos a cabo en este trabajo y al final de este capítulo, en la sección 7.4, estudiaremos las tareas involucradas en la *realización lingüística* de nuestro sistema de NLG.

7.2. Entrada y salida del realizador de superficie

El objetivo del realizador de superficie será transformar la especificación abstracta del documento, producida por la etapa de *microplanning* en texto de superficie. Este texto estará formado por anotaciones dependientes del sistema de presentación y oraciones generadas por la tarea de realización lingüística.

Será responsabilidad de la tarea de realización lingüística producir frases léxica, morfológica y gramaticalmente correctas para las especificaciones de frase contenidas en la especificación del texto. Por otro lado, será la tarea de realización estructural la encargada de estructurar estas frases en el documento final, utilizando para esto las anotaciones necesarios para el sistema de presentación. Está última tarea resulta claramente dependiente del sistema de presentación utilizado. Por ejemplo, si nuestro sistema debe producir un documento LaTeXpara luego ser compilado y generar un PDF, será el realizador estructural el responsable de producir el código necesario para que luego éste se compile apropiadamente. Por ejemplo, en la figura 6.1 podemos observar un documento producido para la especificación del texto de la figura 7.1.

```
\documentclass{article}
\title{Descripción clase de prueba del ejemplo: SymbolTable}

\begin{document}
\maketitle

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

\end{document}
```

Figura 7.1: Texto final en LaTeX

A continuación, en las próximas secciones, estudiaremos las tareas de realización estructural y lingüística realizadas para este trabajo.

7.3. Realización estructural

Esta tarea deberá transformar los constructores lógicos existentes en la especificación del texto en constructores del sistema de presentación. 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; luego estas serán procesadas y presentadas de manera apropiada permitiéndole al lector una correcta visualización. Por ejemplo, para crear una lista en LaTeX, debemos utilizar el entorno *itemize*, para lo cual tenemos que inicializar y terminar el mismo mediante las sentencias `\begin{itemize}` y `\end{itemize}` respectivamente y entre las anteriores utilizar el comando `\item` para indicar los elementos de la lista.

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 las oraciones o frases a generar. Por ejemplo, podemos observar en la figura 7.1 el código LaTeXresultado de la realización de superficie para la especificación de frase introducida en el capítulo anterior (sección 6.1). En la figura 7.2 podemos ver otra realización para la misma especificación de texto pero en lenguaje HTML, donde es posible notar que el texto contenido en ambos ejemplos es exactamente el mismo, diferenciándose únicamente en las anotaciones utilizadas para cada sistema de presentación.

```
<html>
<body>
  <h1>Descripción clase de prueba del ejemplo: SymbolTable</h1>

  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>

</body>
</html>
```

Figura 7.2: Texto final en HTML

A continuación, estudiaremos la tarea de realización estructural para nuestro sistema asumiendo que debemos generar únicamente código LaTeXpara luego ser compilado y convertido a un archivo PDF, resultando esto fácilmente extensible para la generación de código para otros sistemas de presentación. Para esto, tenemos que considerar cómo *mapear* los dos constructores lógicos de nuestra especificación de texto (figura 6.1) en anotaciones o sentencias de LaTeX. Para realizar *TSDocumento* deberemos agregar las etiquetas de encabezado necesarias, especificar el tipo de documento y el título, y delimitar el principio y el final del contenido. Luego deberemos realizar recursivamente los elementos contenidos en el documento (*TSListaItems*, figura 6.3) y ubicarlos adecuadamente en el cuerpo del mismo.

Algoritmo 2 Realización superficie

```
1:  function REALIZAR(TSDocumento doc)
2:      resultado ← "\documentclass{article}"
3:      resultado ← resultado + "\title{" + VERBALIZAR(doc.titulo) + "}"
4:      resultado ← resultado + "\begin{document}"
5:      resultado ← resultado + "\maketitle"
6:      for each desc in doc.descs do
7:          resultado ← resultado + REALIZAR_ITEMS(desc)
8:      end for
9:      resultado ← resultado + "\end{document}"
10:     return resultado
11: end function

12: function REALIZAR_ITEMS(TSListaItems li)
13:     resultado ← VERBALIZAR(li.encabezado)
14:     resultado ← resultado + "Cuando:"
15:     resultado ← resultado + "\begin{itemize}"
16:     for each ef in li.elementos do
17:         resultado ← resultado + "\item{" + VERBALIZAR(ef) + "}"
18:     end for
19:     resultado ← resultado + "\end{itemize}"
20:     return resultado
21: end function
```

En el algoritmo 2 podemos ver el pseudocódigo para la tarea de realización estructural que nos permitirá transformar nuestras especificaciones de texto en código LaTeX. Para esto necesitaremos construir primero el código necesario para el preámbulo: aquí es donde definiremos el título del documento, donde podemos notar la llamada a la función VERBALIZAR; esta será la encargada de llevar a cabo la tarea de realización lingüística y producir una frase en lenguaje natural en base a una especificación de frase dada para el título. Luego construiremos el cuerpo del documento, procesando una por una las

descripciones de las clases de prueba a generar, delegando la responsabilidad de la realización de cada una de estas en la función REALIZAR_ITEMS. Ésta agregará el código necesario para construir la lista de ítems, requiriendo también para este caso la realización lingüística de las especificaciones de frases presentes en cada *TSListaItems*.

Para completar la realización de superficie será necesario definir el comportamiento de la función VERBALIZAR antes mencionada. Ésta realizará la tarea de realización lingüística y en la próxima sección nos encargaremos especificar la misma.

7.4. Realización lingüística

La realización lingüística desempeñará su tarea al nivel de la oración. La misión de ésta será transformar las especificaciones de frases producidas por la etapa anterior en oraciones bien formadas; entendiendo por oraciones bien formadas a aquellas que cumplen con las reglas gramaticales de la lengua en la cual se encontrará redactado el documento final (castellana en nuestro caso). La realización lingüística, entonces, consistirá en aplicar alguna caracterización de estas reglas a cada especificación de frase a fin de producir un texto que sea sintáctica, ortográfica y gramaticalmente correcto.

En primer lugar, para que el texto producido sea ortográficamente correcto, asumiremos que las designaciones provistas por el usuario son ortográficamente correctas y, por nuestro lado, nos comprometeremos a que las frases generadas por nuestro sistema también lo sean. Además de esto, deberemos encargarnos de que cada oración comience con mayúscula y termine con un signo de puntuación.

En el capítulo 4.6 introdujimos algunas consideraciones sobre concordancia gramatical, las cuales deberán ser tenidas en cuenta por nuestra tarea de realización lingüística. A continuación, describiremos las reglas de concordancia mencionadas, para luego, al finalizar el capítulo, definir la realización lingüística para los distintos elementos de nuestra especificación de frase.

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 especificador podría no incluir el artículo en el texto utilizado en una designación. Por ejemplo, podría haber omitido el artículo en una de las designaciones utilizadas en los capítulos anteriores (figura 2.4) y en lugar de designar:

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

haber designado:

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

Luego, para verbalizar la expresión $s? \in \operatorname{dom} st$, por ejemplo, necesitaremos incluir el artículo en el caso que no se encuentre presente:

$$verb(s? \in \operatorname{dom} st) \rightarrow \textit{“}\textbf{\textit{El}}\textit{ símbolo a buscar pertenece a los símbolos cargados en la tabla de símbolos”.}$$

Será entonces nuestro sistema el encargado de agregar este artículo en el caso de ser necesario[2], asegurándose de que el mismo concuerde con el sustantivo. Para esto deberemos ser capaces de identificar el género y número del sustantivo[3].

Concordancia entre sujeto y atributo de verbo copulativo. Establece que el atributo de un verbo copulativo (ser, estar y parecer) debe concordar en género y número con el núcleo del sujeto de la oración. Veamos por ejemplo el siguiente caso incluido en la definición presentada en la sección 4.5:

```
verb' (exp1 ⊂ exp2)
    | (gen == M && num == S) = verb(exp1) +
                               "esta incluido en" +
                               verb(exp2)
    | (gen == M && num == P) = verb(exp1) +
                               "estan incluidos en" +
                               verb(exp2)
    | (gen == F && num == S) = verb(exp1) +
                               "esta incluida en" +
                               verb(exp2)
    | (gen == F && num == P) = verb(exp1) +
                               "estan incluidas en" +
                               verb(exp2)
    where gen = genero(exp1)
          num = numero(exp1)
```

Donde podemos ver que tenemos el verbo copulativo *“estar”* acompañado del atributo *“incluido”*. En este caso deberemos escoger el género y número 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. Como vimos durante el análisis de requerimientos en el capítulo 4, las frases a generar por nuestro sistema se encontrarán siempre en tercera persona por lo cual no será necesario preocuparse por la

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

[3]Para esto, haremos uso de un analizador morfológico que determinará la forma, clase o categoría gramatical de cada palabra de una oración. En el capítulo 8 nos explayaremos más en detalle sobre el analizador morfológico utilizado para este trabajo.

concordancia de persona entre sujeto y verbo de la oración. Por lo tanto, nuestro realizador deberá asegurarse únicamente de que el número del verbo y el sujeto concuerden. Veamos, por ejemplo, el siguiente caso también incluido en la definición presentada en la sección 4.5, que el verbo *"pertenecer"* deberá concordar en número con el sujeto (en este caso exp_1).

```
verb' (exp1 ∈ exp2)
    | (num == S) = verb(exp1) + "pertenece a" + verb(exp2)
    | (num == P) = verb(exp1) + "pertenecen a" + verb(exp2)
    where num = numero(exp1)
```

Otra cuestión a observar, además de las reglas de concordancia ya vistas, es que nuestro sistema deberá generar casi exclusivamente oraciones bimembres ordenadas como:

$$\textbf{\textit{sujeto}} + \textbf{\textit{verbo}} + \textbf{\textit{objeto}}$$

Nuestro realizador deberá siempre respetar este orden de palabras para realizar los elementos de tipo *Oracion* de nuestra especificación de frase.

Debemos aclarar que en la definición introducida en la sección 4.5 podemos observar oraciones en las cuales el orden de palabras difiere del mencionado anteriormente, por ejemplo, la frase:

"no hay elementos en la tabla de símbolos"

cuyo orden es:

$$\textit{verbo} + \textit{sujeto} + \textit{objeto}$$

Para estos casos excepcionales en que el orden svo no se cumple, utilizaremos *ElementosYuxtapuestos* combinando frases enlatadas y nominales para el modelado de los mismos. Por ejemplo, modelaremos la frase presentada anteriormente mediante un *ElementosYuxtapuestos* entre una *FraseEnlatada* con el texto "no hay elementos en" y la especificación de frase correspondiente al objeto al que hace referencia, para el ejemplo anterior será una *FraseNominal* utilizada para modelar el texto "la tabla de símbolos" proveniente de las designaciones de la especificación.

En base a los aspectos mencionados anteriormente analizaremos la tarea de realización lingüística para cada elemento de nuestra especificación de frase. Esta tarea será recursiva sobre la estructura de las especificaciones antes mencionadas. Para ilustrar el comportamiento de la misma presentaremos una definición en pseudocódigo para cada posible elemento de una especificación de frase. Llamaremos VERBALIZAR a la función encargada de llevar a cabo

esta tarea; será la encargada de generar una oración sintáctica y gramaticalmente correcta en base a una especificación de frase dada. 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 (en particular, deberemos asegurarnos de que la primera letra de cada oración sea mayúscula y de agregar un signo de puntuación al final de cada una de éstas).

A continuación analizaremos la realización lingüística de cada uno de los elementos que pueden formar parte de una especificación de frase:

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

Algoritmo 3 Realización lingüística *FraseEnlatada*

```
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[4] y respetando el orden en que se encuentren.

Algoritmo 4 Realización lingüística *ElementosYuxtapuestos*

```
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.

[4]Suponemos que la función CONCAT además de concatenar los elementos agrega un espacio entre cada uno de éstos.

Algoritmo 5 Realización lingüística *ElementosCoordinados*

```
1: function VERBALIZAR(ElementosCoordinados elem)
2:     for each e in elem.elementos do
3:         if ES_PRIMER_ELEMENTO(e, elem) then
4:             resultado ← VERBALIZAR(e)
5:         else if ES_ULTIMO_ELEMENTO(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*, *nucleo* y verbalizar recursivamente *complemento* agregando los espacios correspondientes entre medio.

Algoritmo 6 Realización lingüística *FraseNominal*

```
1: function VERBALIZAR(FraseNominal frase)
2:     nucleo ← frase.nucleo
3:     complemento ← VERBALIZAR(frase.complemento)
4:     if ES_NOMBRE(nucleo) then
5:         resultado ← CONCAT(nucleo, complemento)
6:     else
7:         determinante ← DETERMINAR_ARTICULO(frase)
8:         resultado ← CONCAT(determinante, nucleo, complemento)
9:     end if
10:    return resultado
11: end function
```

Observemos que se deberá verbalizar recursivamente el complemento de la frase (éste probablemente sea una frase enlatada o una yuxtaposición de las mismas). Finalmente se construye un texto con el *determinante*, *núcleo* y la verbalización del *complemento*, respetando el orden antes mencionado.

La función DETERMINAR_ARTICULO deberá recuperar el determinante apropiado para la frase; en el caso en que la frase ya contenga un determinante tendrá que devolver el mismo y en caso contrario (de ser necesario) deberá determinar el artículo indicado según número y género del núcleo de la frase. Es razonable suponer que el sustantivo utilizado en el texto de una designación, identifica algo conocido y por lo tanto deberá ser un artículo definido el que lo preceda. Los artículos definidos son aquellos que hablan de algo conocido y que se puede identificar, en particular son artículos definidos: el/lo/las/los.

Nuestro sistema podrá entonces tener este conjunto de artículos previamente indexado por número y género de la siguiente manera:

Masculino		**Femenino**	
Singular	**Plural**	**Singular**	**Plural**
el	los	la	las

y de esta forma determinar según el número y género del sujeto presente en el texto de la designación el artículo apropiado para la misma.

La función ES_NOMBRE deberá hacer uso de un analizador morfológico (nos explayaremos sobre este tema en el capítulo 8) para determinar si el núcleo de la frase nominal es un nombre propio. En ese caso no deberíamos agregar un artículo.

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 el elemento *FraseVerbal* que corresponde al predicado del mismo.

Oracion. Este es el caso más interesante para nuestro verbalizador. Deberemos generar una oración correcta en base a las reglas de concordancia gramatical antes mencionadas.

Algoritmo 7 Realización lingüística Oracion

```
1: function VERBALIZAR(Oracion oracion)
2:     sujeto ← VERBALIZAR(oracion.sujeto)
3:     fverbal ← oracion.predicado
4:     verbo ← DETERMINAR_VERBO(fverbal.verbo, oracion.sujeto)
5:     atributo ← DETERMINAR_ATRIBUTO(fverbal.atributo, oracion.sujeto)
6:     complemento ← VERBALIZAR(fverbal.complemento)
7:     resultado ← CONCAT(sujeto, verbo, atributo, complemento)
8:     return resultado
9: end function
```

A fin de contemplar estas reglas, le prestaremos especial atención al verbo y al atributo de la frase verbal correspondiente al predicado de la oración, verbalizando recursivamente tanto el sujeto (generalmente una frase nominal producto de una designación) como el complemento de la frase verbal. Veremos en el capítulo 8 cómo nuestro sistema se encargará de agregar información morfológica a las especificaciones; la cual será necesaria en esta etapa para, por ejemplo, conocer el número y género del núcleo del sujeto con el fin de determinar la conjugación apropiada para el verbo y el atributo a utilizar.

La función DETERMINAR_VERBO será la encargada de conjugar el verbo (que en la especificación de frase se encuentra en su forma canónica, es decir, el infinitivo del verbo) de manera que concuerde en número y persona con el

sujeto, mientras que la función DETERMINAR_ATRIBUTO tendrá la tarea de determinar el atributo de forma que concuerde en número y género también con el sujeto. Finalmente se unirán los constituyentes en el orden mencionado previamente: *sujeto - verbo - predicado*.

Tanto para determinar la conjugación del verbo como el atributo a utilizar deberemos contar con un léxico en el que cada palabra se encuentre indexada según su forma canónica, número y género, de forma tal que nos permita obtener el verbo o atributo correctamente conjugado en base a las características anteriores. En nuestro caso, los verbos y atributos utilizados por el sistema dependen del mapeo entre operadores y verbalizaciones introducido en la sección 6.3. Es decir, conocemos los verbos y atributos que utilizaremos en las oraciones[5] de nuestro sistema de antemano. De esta forma, podemos establecer la conjugación de los verbos utilizados en las oraciones de nuestro sistema:

Verbo	**Singular**	**Plural**
Estar	*está*	*están*
Pertenecer	*pertenece*	*pertenecen*
Ser	*es*	*son*

Por otro lado, usaremos las siguientes relaciones para decidir qué atributo debemos utilizar:

	Masculino		**Femenino**	
Atributo	**Singular**	**Plural**	**Singular**	**Plural**
Igual	*igual*	*iguales*	*igual*	*iguales*
Incluido	*incluido*	*incluidos*	*incluida*	*incluidas*

Por ejemplo, para la realización lingüística de la oración presentada en la figura 6.4 del capítulo anterior, de la cual sabremos que el núcleo del sujeto es singular, y el género del mismo es masculino, nuestro realizador deberá determinar el verbo adecuado de la siguiente manera:

$$\text{DETERMINAR\_VERBO}(\textit{“pertenecer”}, \textit{“símbolo”}) \rightarrow \textit{“pertenece”}$$

En este caso, la función DETERMINAR_VERBO deberá escoger la opción correspondiente en base a un análisis morfológico de su segundo argumento (retomaremos sobre este tema en el capítulo 8). De forma similar deberá comportarse la determinación de atributo en caso de ser de ser necesaria.

[5]En este caso hacemos referencias a oraciones modeladas mediante el elemento *Oracion*. Pueden aparecer otros verbos y atributos en los textos generados, pero serán producto de frases enlatadas, para las cuales no realizaremos ningún procesamiento para satisfacer la concordancia gramatical; supondremos que estas frases son gramaticalmente correctas.

7.5. Resumen del capítulo

En este capítulo vimos cómo finalmente, a partir de una especificación del documento producida por el *microplanner*, nuestro sistema, a través de las tareas de realización estructural y realización lingüística es capaz de producir el documento final formado por frases en lenguaje natural, símbolos de puntuación y anotaciones necesarias para el sistema de presentación. En el próximo capítulo nos concentraremos en estudiar los aspectos más relevantes de la implementación realizada para este trabajo.

Capítulo 8

Implementación

En éste capítulo introduciremos los detalles más relevantes del desarrollo realizado para este trabajo. Comenzaremos mostrando la integración de nuestro sistema de NLG con *Fastest*, viendo el modo de uso y los nuevos comandos introducidos para la generación de descripciones en lenguaje natural. Luego presentaremos los aspectos más destacados de nuestra implementación presentes en cada una de las etapas del *pipeline*.

8.1. Integración con *Fastest*

La implementación realizada para este trabajo se encuentra incluida en la última versión de *Fastest*, cuyo código está disponible públicamente[1]. La mayor parte del código referente a este trabajo la podremos encontrar dentro del paquete *nlg*.

Requisitos

Para garantizar el correcto funcionamiento de Fastest y nuestro sistema de NLG, deberemos cumplir con los siguientes requerimientos:

- *Fastest 1.6 o superior*: la distribución de éste incluye un pequeño manual de uso.
- *Java SE Runtime Environment 1.6 o superior*: requerido para el correcto funcionamiento de *Fastest*.
- *SWI_Prolog*[2]: también requerido para el correcto funcionamiento de Fastest.
- *FreeLing 3.1*[3].: suite de análisis de lenguajes, necesaria para el módulo de NLG de *Fastest*

[1] http://github.com/rosacris/fastest

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

[3] http://nlp.lsi.upc.edu/freeling/. Hemos creado un pequeño tutorial para instalación de *FreeLing* en *Ubuntu*; disponible en http://gist.github.com/juliandt/

8.1.1. Modo de uso

Como resultado de este trabajo introdujimos un nuevo comando en *Fastest*, el comando `showdesc`. Podremos usar el mismo para generar la descripción de una o más clases de prueba, pasándole el título deseado para el documento junto con el o los nombres de las clases de prueba a traducir. También es posible pasar como argumento el nombre de una operación de la especificación y nuestro sistema buscará todas las clases de prueba generadas para esa operación produciendo luego las descripciones para las mismas. En la figura, 8.1 podemos ver la ayuda de *Fastest* para el comando anterior.

```
showdesc [-t <title>] [<sch_name>]
        Generates natural language description for specified schemas.
```

Figura 8.1: Ayuda para el comando showdesc en *Fastest*

Veremos a continuación un ejemplo de uso del comando `showdesc`, mediante un ejemplo similar al introducido en la sección 2.1.5 (donde ilustramos el funcionamiento general de *Fastest*):

```
Fastest version 1.6, (C) 2013, Maximiliano Cristiá
Loading pruning rewrite rules...
Loading pruning theorems...
Fastest> loadspec symbolTable.tex
Loading specification..
Specification loaded.
Fastest> selop LookUp
Fastest> addtactic LookUp SP \in s? \in \dom st
Fastest> genalltt
Generating test tree for 'LookUp' operation.
Fastest> showdesc -t "Descripción clase de prueba: LookUp_SP_1" LookUp_SP_1

\documentclass{article}
\title{Descripción clase de prueba: LookUp_SP_1}

\begin{document}
\maketitle

LookUp_SP_1: Se intenta buscar un simbolo en la tabla
Cuando:
\begin{itemize}
 \item{El simbolo a buscar pertenece a los simbolos cargados en la tabla.}
 \item{El simbolo a buscar es el único elemento en los simbolos cargados en la tabla.}
\end{itemize}

\end{document}
```

Figura 8.2: Comandos para generación de descripciones en *Fastest*

Por otro lado, para poder hacer uso de las designaciones presentes en la

especificación hemos definido un nuevo entorno y un nuevo comando para facilitarnos el parseo de las mismas. En la figura 8.3 podemos observar el código necesario que el especificador deberá incluir junto a la especificación.

```
\newenvironment{designations}[1]
  {\begin{leftbar}
    \begin{list}{}{\setlength{\labelsep}{0cm}
                   \setlength{\labelwidth}{0cm}
                   \setlength{\listparindent}{0cm}
                   \setlength{\rightmargin}{\leftmargin}}}
  {\end{list}\end{leftbar}}

\newcommand{\desig}[2]{\item #1 $\approx #2$}
```

Figura 8.3: Comandos necesarios para la definición de designaciones en la especificación

Luego deberá introducir las designaciones haciendo uso del nuevo comando definido. En la figura 8.4 podemos ver un bloque de designaciones haciendo uso del comando presentado en la figura 8.3. Podemos observar que es posible pasarle un nombre de esquema como parámetro al entorno `designations` para delimitar el *scope* de las variables designadas. De no pasarle un nombre de esquema como parámetro el sistema considerará que el scope del bloque de designaciones es toda la especificación (utilizamos esta alternativa por ejemplo para designar nombres de esquemas, tipos básicos, etc.).

```
\begin{designations}{}
  \desig{se intenta buscar un simbolo en la tabla}{LookUp}
  \desig{se intenta actualizar la tabla de simbolos}{Update}
\end{designations}

\begin{designations}{LookUp}
  \desig{el simbolo a buscar}{s?}
  \desig{la tabla de simbolos}{st}
  \desig{simbolos cargados en la tabla}{\dom st}
\end{designations}
```

Figura 8.4: Ejemplo designaciones *SymbolTable*

8.2. Detalles de la implementación

En esta sección detallaremos los aspectos más importantes de la implementación realizada. Para la misma respetamos la arquitectura y desarrollo realizado a lo largo de este trabajo. Sin embargo podremos observar algunas diferencias menores, como por ejemplo, la necesidad de modificar los nombres

de las entidades intermedias utilizadas entre las distintas etapas del *pipeline* para respetar las convenciones de estilo utilizadas en Fastest.

A continuación describiremos la función de los componentes más importantes usados en cada etapa de nuestro sistema.

8.2.1. *Document Planner*

El módulo *DocumentPlanner* será el encargado de llevar a cabo las tareas de determinación de contenido y estructuración del documento detalladas en el capítulo 5. En la figura 8.5 podemos observar los componentes más importantes involucrados en esta etapa, cuya función describiremos a continuación.

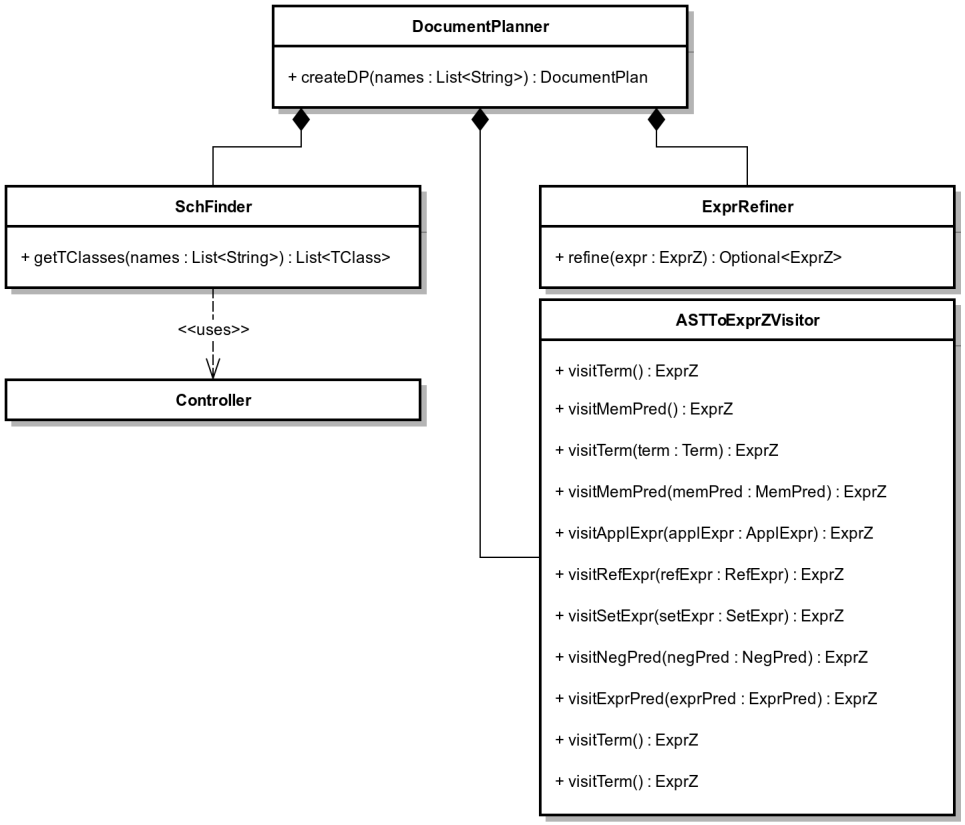

Figura 8.5: Diagrama clases para *DocumentPlanner*

DocumentPlanner. Es el módulo encargado de crear un *document plan* a partir de la entrada de nuestro sistema. Construye los mensajes y estructuras intermedias de éste, delegando las tareas de determinación de contenido en los módulos que describiremos a continuación.

SchFinder. La tarea de selección detallada en la sección 5.4 será desarrollada por este módulo, que será el encargado de recuperar el conjunto de clases de prueba indicadas. El mismo posee una referencia al módulo *Controller*[4] de Fastest que le permitirá identificar y recuperar las clases de prueba necesarias.

ExprRefiner. Es la *fachada* [GHJV95] encargada de delegar los distintos procesamientos a realizar sobre las expresiones a fin de desarrollar las tareas de eliminación de tautologías y reducción de expresiones estudiadas en la sección 5.4. Podemos observar que utilizamos la clase `Optional` de Java 8 para modelar el hecho de que el método `refine()` podría no devolver un valor, por ejemplo en el caso que la expresión procesada sea una tautología y no deba ser incluida en el *document plan*. En el capítulo 9 propondremos como trabajo a realizar, la implementación de nuevas tareas de razonamiento con los datos; será en esta clase en la que deberemos agregar las el comportamiento necesario para contemplar dichas tareas (cada una de éstas debería estar encapsulada un un módulo separado y el módulo ExprRefiner debería tener una referencia a los mismos e implementar las llamadas a los métodos de correspondientes).

Fastest utiliza el *framework* CZT[5] que integra un conjunto de herramientas para trabajar con el lenguaje de especificación Z. La forma en la que CZT modela las expresiones de Z nos resultó compleja para este trabajo, por lo que decidimos transformar las expresiones contempladas dentro del alcance de este trabajo a un modelo más simple (lo que simplificó la implementación del algoritmo de lexicalización, por ejemplo). En la figura 8.6 podemos observar la jerarquía de clases de las expresiones utilizadas en este trabajo para modelar las expresiones Z.

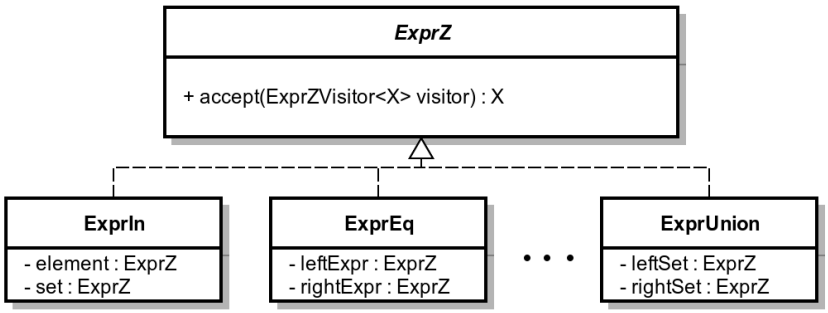

Figura 8.6: Diagrama jerarquía de clases para ExprZ

ASTToExprZVisitor. Este módulo será el responsable de la transformación

[4] *Controller* es el módulo encargado de mantener las referencias a elementos de la especificación, arboles de prueba, etc.

[5] http://czt.sourceforge.net/

entre el modelo de CZT y el utilizado por este trabajo presentado en la figura 8.6.

8.2.2. *Microplanner*

En esta sección presentaremos los componentes encargados de la tarea de *microplanning* detallada en el capítulo 6. En la figura 8.7 podemos observar los módulos involucrados en la implementación de esta tarea.

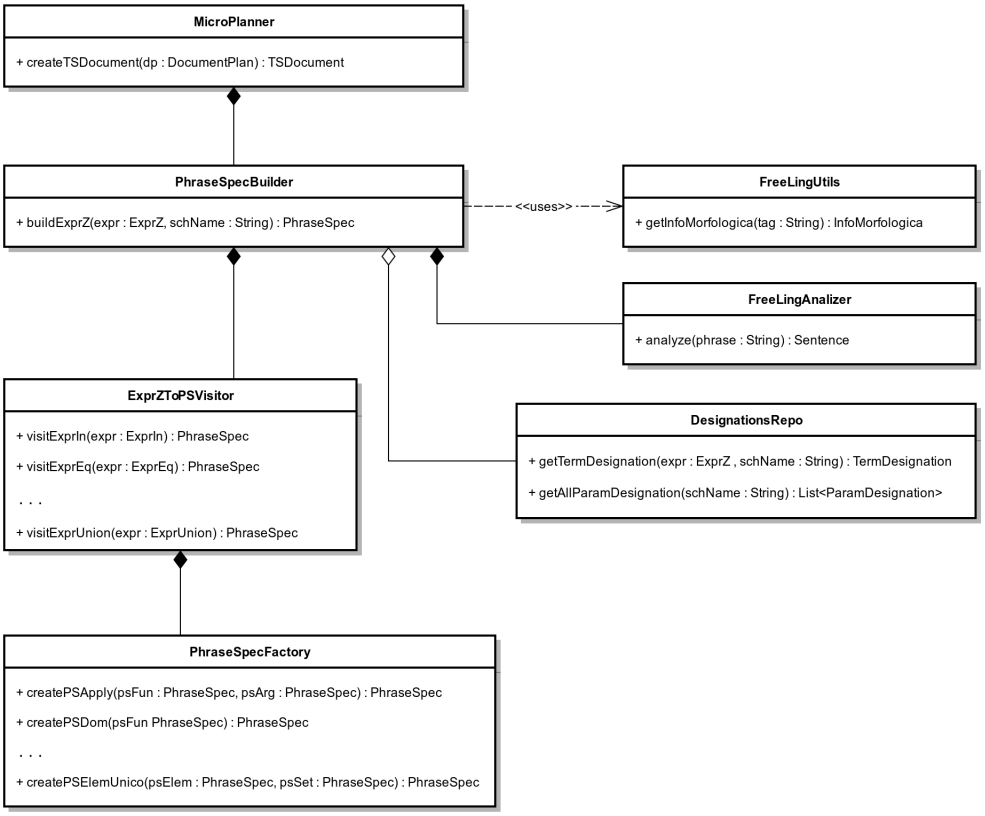

Figura 8.7: Diagrama clases para *Microplanner*

MicroPlanner. Es el módulo principal de esta etapa, encargado de construir la especificación del documento a partir del *document plan* generado por la etapa anterior. Éste delega la tarea de lexicalización en el módulo *PhraseSpecBuilder* que presentaremos a continuación.

PhraseBuilder. Este módulo implementa la tarea de lexicalización detallada en el capítulo 6, la función del mismo será construir una especificación de frase a partir de una expresión Z. Para realizar esta tarea necesitaremos recorrer el modelo utilizado para las expresiones Z. Encapsulamos este trabajo en el

módulo *ExprZToPSVisitor*, encargado del análisis de casos para las distintas expresiones y sus posibles combinaciones de acuerdo a la definición presentada en la sección 6.3.

ExprZToPSVisitor. Encapsula las reglas de lexicalización para todas las posibles expresiones y combinaciones de las mismas. Implementa la función auxiliar LEXICALIZACIÓN'() utilizada en el bosquejo de la figura 1. Utilizamos el patrón de diseño *visitor* [GHJV95] para iterar sobre la estructura de las distintas expresiones Z modeladas por nuestro sistema.

PhraseSpecFactory. Construir y componer las especificaciones de frase de nuestro sistema resulta una tarea relativamente compleja. Es por esto que encapsulamos esta funcionalidad en el módulo *PhraseSpecFactory* encargado de construir apropiadamente las distintas especificaciones de frase de nuestro sistema. Por ejemplo: `createPSElemUnico()` tomará dos especificaciones de frase y las compondrá para generar una nueva especificación para la frase *"... es el único elemento de ..."*, siendo las dos especificaciones mencionadas anteriormente las encargadas de modelar el texto que precede y antecede respectivamente a la anterior.

FreeLingAnalizer. Utilizamos el analizador morfosintáctico *FreeLing* para obtener las características morfológicas de los distintos constituyentes de las frases utilizadas en las designaciones. Este módulo tendrá la tarea de interactuar con las librerías provistas por la herramienta proveyéndonos de la información necesaria (como el núcleo de la frase, su género, número, etc.) para poder construir una especificación de frase a partir de una designación.

FreeLingUtils. Contiene algunos métodos útiles para el trabajo con *FreeLing*. Por ejemplo, *FreeLing* genera una estructura en la que etiqueta cada palabra con anotaciones morfosintácticas. El método `getInfoMorfologica()` procesa estas anotaciones produciendo una estructura más sencilla con la que trabajará nuestro sistema.

DesignationsRepo. Es necesario para el algoritmo de lexicalización saber si una expresión se encuentra designada y, en ese caso, necesitará saber cúal es su designación. Será a través de este módulo que podrá acceder a las designaciones presentes en la especificación. El mismo se inicializa al cargar la especificación en Fastest.

8.2.3. *Surface Realizer*

Finalmente, el módulo *SurfaceRealizer* será el encargado de generar el texto de superficie para el documento final en base a una especificación del mismo. En esta etapa se realizan las tareas de realización estructural y lingüística presentadas en el capítulo 7. En la figura 8.8 podemos ver los componentes más

relevantes involucrados en esta etapa, cuya función describiremos a continuación.

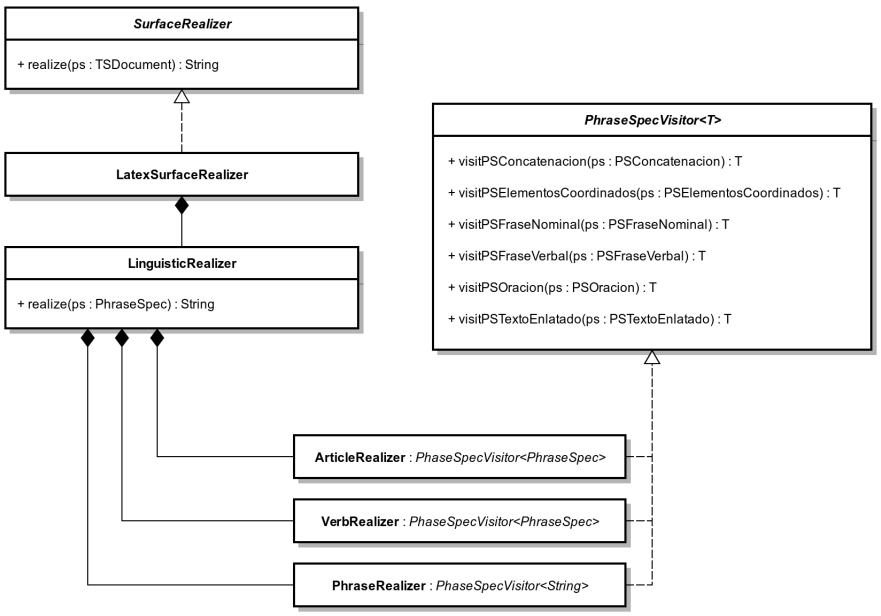

Figura 8.8: Diagrama clases para *Surface Realizer*

SurfaceRealizer. Como mencionamos en la sección 7 la tarea de realización estructural resulta dependiente del sistema de presentación utilizado. Es por esto que definimos la interfaz abstracta *SurfaceRealizer* y una implementación especifica encargada de producir código LaTeX apropiado para el documento final. El método principal de esta clase toma una especificación del documento final y tiene la misión de producir el texto en lenguaje natural esperado, para esto deberá hacer uso del *LinguisticRealizer* para lograr la realización lingüística de cada una de las especificaciones de frase presentes en la especificación de documento dada.

LinguisticRealizer. Es el módulo encargado de la realización lingüística. Es responsabilidad de este módulo asegurarse de que las frases del texto generado respeten las reglas gramaticales introducidas en el capítulo 7.4. Éste hace uso de los módulos *ArticleRealizer* y *VerbRealizer* para resolver las cuestiones de concordancia gramatical, delegando la generación del texto final en el módulo *PhraseRealizer*.

PhraseSpecVisitor. Utilizamos el patrón *visitor* para iterar sobre la estructura de las especificaciones de frase. A continuación veremos la función de las tres implementaciones más importantes de esta interfaz.

ArticleRealizer. Es el módulo encargado de recorrer la estructura de una especificación de frase y escoger el articulo apropiado (de ser necesario en una frase nominal) de forma tal que concuerde según las reglas gramaticales introducidas previamente. El artículo será agregado sobre la misma estructura utilizada para la especificación de frase.

VerbRealizer. Este módulo tiene la responsabilidad de conjugar el verbo de una oración (establecido en infinitivo por el *microplanner*) y escoger el atributo apropiado en caso de ser un verbo copulativo, teniendo en cuenta los aspectos gramaticales detallados en la sección 7.4. Al igual que con *ArticleRealizer*, reutilizaremos la estructura utilizada para la especificación de frases para establecer el verbo correcto y atributo (reemplazando el verbo infinitivo por el verbo correctamente conjugado y el atributo previamente establecido por el atributo correspondiente).

PhraseRealizer. Será el módulo finalmente encargado de recorrer la estructura de una especificación de frase y generar el texto final correspondiente a la misma. En esta etapa casi todo el trabajo fue realizado. La función de este módulo será extraer el texto contenido en la especificación de frase con el fin de generar la frase final respetando el orden establecido e introduciendo los espacios y signos de puntuación necesarios. Es importante aclarar que resulta necesario que ejecutemos en primer instancia las tareas realizadas por *PhraseSpecVisitor* y *VerbRealizer* ya que *PhraseRealizer* asumirá que ya fueron resueltos previamente todos los aspectos gramaticales.

8.3. Resumen del capítulo

En este capítulo presentamos el prototipo desarrollado como resultado de este trabajo. Mostramos su integración con Fastest, presentando los nuevos comandos introducidos para la generación de descripciones de clases de prueba y describimos su modo de uso. Detallamos también la forma en la que el especificador debe escribir las designaciones para que el sistema pueda hacer uso de ellas. Finalmente comentamos los aspectos más interesantes de la implementación realizada junto con algunas recomendaciones para futuras modificaciones.

Capítulo 9

Conclusiones y trabajos futuros

En este trabajo hemos presentado y desarrollado una solución para la generación automática de descripciones en lenguaje natural de clases de prueba generadas por el TTF. Para esto, fue necesario utilizar técnicas de generación de lenguaje natural. En particular, seguimos la metodología propuesta por Reiter y Dale [RD00].

En primera instancia, recolectamos y analizamos un corpus con textos de ejemplo. A partir de éste extrajimos los requerimientos más importantes para el desarrollo de nuestro sistema. Luego, estudiamos en detalle las tareas más importantes que deben ser llevadas a cabo por nuestro sistema. En el capítulo 5 hicimos especial hincapié en la determinación de contenido y en cómo el razonamiento con los datos puede mejorar notablemente la calidad de las descripciones generadas por nuestro sistema. En el capítulo 6 utilizamos el conjunto de reglas de traducción obtenido como resultado del análisis del corpus para definir la tarea de lexicalización de expresiones Z, encargada de escoger qué palabras y constructores sintácticos utilizar para comunicar la información requerida. Estudiamos, además, el importante rol que cumplen las designaciones para este trabajo y cómo nuestra tarea de lexicalización hace uso de la información contenida en las mismas. En el capítulo 7 nos encargamos fundamentalmente de estudiar los aspectos necesarios para, finalmente, generar texto sintáctica, ortográfica y gramaticalmente correcto.

Como resultado de este trabajo, hemos implementado un prototipo en base al desarrollo realizado, utilizando para esto un diseño flexible capaz de admitir futuras posibilidades de evolución. El sistema desarrollado trabaja principalmente con la información contenida en las clases de prueba, haciendo un especial uso de las designaciones que acompañan a la especificación. Estas designaciones resultan una fuente fundamental de información que le permite a nuestro sistema generar las porciones de texto dependientes del dominio de aplicación y de esta forma lograr un sistema de generación de descripciones

en lenguaje natural independiente del dominio de aplicación y de la cantidad de clases y casos de pruebas.

El trabajo descripto en esta tesis es el primero en el cual se aplican técnicas del estado del arte en la generación de lenguaje natural al problema de accesibilidad de las especificaciones formales. Esto continua con la línea de investigación iniciada por Cristiá y Plüss [CP10] en busca de métodos y herramientas que traduzcan especificaciones formales de sistemas reales en documentos accesibles para expertos en el dominio sin formación en métodos formales específicos.

Trabajos futuros

El trabajo realizado esta abierto a nuevos aportes. A continuación detallaremos algunos de los aportes que creemos que podrían realizarse como trabajo futuro.

Una continuación importante para nuestro trabajo sería la realización de una evaluación de los textos generados por el sistema. Para esto deberíamos juntar un grupo de personas capaz de leer Z y proveerles un conjunto de clases de prueba acompañado de sus descripciones en lenguaje natural generadas por nuestro prototipo. Estas personas deberán evaluar las mismas en términos de exactitud, fluidez del texto, etc. Podríamos además incluir descripciones traducidas manualmente por expertos en el dominio de aplicación para luego comparar los resultados de las descripciones generadas por el sistema contra descripciones redactadas por un experto.

El sistema desarrollado no cubre la totalidad de las expresiones y operadores de Z; dentro del alcance de este trabajo establecimos un subconjunto de éstos en el que nos enfocamos. Una continuación natural de este trabajo sería la extensión del conjunto de expresiones soportadas por nuestro sistema. Para esto debería expandirse el corpus, contemplando casos para los operadores a introducir y luego volver a realizar un análisis del mismo extrayendo nuevas reglas para describir las nuevas expresiones en lenguaje natural.

También es posible profundizar el trabajo sobre los datos de entrada e incluir nuevas tareas de razonamiento sobre los mismos en la etapa de *document planning*. Esto le permitiría a nuestro sistema mejorar la calidad de los textos producidos. Por ejemplo, se podría trabajar sobre las expresiones redundantes que pueden aparecer en una clase de prueba.

Como mencionamos en el capítulo 6, nuestro sistema no realiza tareas de agregación y generación de expresiones de referencia. Una extensión natural sería la de incorporar estas tareas como parte del *document planning*.

El sistema desarrollado actualmente es capaz de producir únicamente textos en castellano. Un aporte importante sería expandir el mismo para permitir la generación de textos en otras lenguas. Para generar textos en inglés, por ejemplo, existen realizadores lingüísticos de código abierto que podrían ser uti-

lizados en este desarrollo. En este caso tendríamos que implementar un nuevo módulo de *microplanning* de modo que sea capaz de generar las estructuras utilizadas por el realizador lingüístico a utilizar.

Apéndice A

Corpus de descripciones

Para construir el corpus utilizado para este trabajo utilizamos 5 especificaciones de distintos dominios de aplicación con sus respectivos casos y clases de prueba generados por Fastest. De estas clases de prueba escogimos las más interesantes para traducir acabando con un total de 98 clases de prueba.

A continuación presentaremos 3 de los ejemplos recolectados que forman parte del corpus utilizado y resultan representativos del total de ejemplos recolectados. En éstos incluimos tanto la especificación utilizada para generar las clases de prueba a traducir, como también las designaciones necesarias para poder describir las clases de prueba seleccionadas y las descripciones de las mismas.

Ejemplo: *Symbol Table*

Especificación y designaciones

$[SYM, VAL]$

$REPORT ::= ok \mid symbolNotPresent$

- Tabla de símbolos $\approx st$
- Símbolos cargados en la tabla $\approx dom\ st$
- Valor de $x \approx st\ x$

ST

$st : SYM \pfun VAL$

- Intenta actualizar el valor de un símbolo $\approx Update?$

- Símbolo a actualizar $\approx s?$
- Valor a actualizar $\approx v?$

Update

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

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

- Símbolo a buscar $\approx s?$
- Valor del símbolo buscado $\approx v?$

LookUpOk

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

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

LookUpE

ΞST
$s? : SYM$
$rep! : REPORT$

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

- Intenta buscar el valor de un símbolo $\approx LookUp?$

$LookUp == LookUpOk \lor LookUpE$

- Símbolo a eliminar $\approx s?$

DeleteOk

ΔST
$s? : SYM$
$rep! : REPORT$

$s? \in \operatorname{dom} st$
$st' = \{s?\} \lhd st$
$rep! = ok$

- Intenta eliminar un símbolo de la tabla $\approx$ *Delete*?

$DeleteE == LookUpE$

$Delete == DeleteOk \lor DeleteE$

Clases de prueba y descripciones

LookUp_SP_1

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

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

LookUp_ SP_ 1

Se intenta buscar el valor de un símbolo, cuando:
- El símbolo a buscar pertenece a los símbolos cargados en la tabla.
- El símbolo a buscar es el único elemento de los símbolos cargados en la tabla.

LookUp_SP_4

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

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

LookUp_ SP_ 4

Se intenta buscar el valor de un símbolo, cuando:
- El símbolo a buscar no pertenece a los símbolos cargados en la tabla.
- Los símbolos cargados en la tabla no son iguales al conjunto formado por el símbolo a buscar.
- El símbolo a buscar pertenece a los símbolos cargados en la tabla.

Update_SP_1

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

$$st = \{\}$$
$$\{s? \mapsto v?\} = \{\}$$

Update_ SP_ 1

Se intenta actualizar el valor de un símbolo, cuando:
- No hay ningún elemento en la tabla de símbolos.
- No hay elementos en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.

Update_SP_2

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

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

Update_ SP_ 2

Se intenta actualizar el valor de un símbolo, cuando:
- No hay ningún elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.

Update_SP_3

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

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

Update_ SP_ 3

Se intenta actualizar el valor de un símbolo, cuando:
- Existe al menos un elemento en la tabla de símbolos.
- No hay elementos en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.

Update_SP_4

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

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

Update_ SP_ 4

Se intenta actualizar el valor de un símbolo, cuando:
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.
- El símbolo a actualizar pertenece a los símbolos cargados en la tabla.

Update_SP_5

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

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

Update_ SP_ 5

Se intenta actualizar el valor de un símbolo, cuando:
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.
- El símbolo a buscar pertenece a los símbolos cargados en la tabla.

Update_SP_6

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

$$st \neq \{\}$$
$$\{s? \mapsto v?\} \neq \{\}$$
$$(\mathrm{dom}\ st \cap \mathrm{dom}\{s? \mapsto v?\}) = \{\}$$

Update_ SP_ 6

Se intenta actualizar el valor de un símbolo, cuando:
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por el par ordenado por: el símbolo a actualizar y el valor a actualizar.
- Los símbolos cargados en la tabla están incluidos en el conjunto formado por el símbolo a actualizar.

Delete_SP_2

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

$$s? \in \mathrm{dom}\ st$$
$$st \neq \{\}$$
$$\{s?\} = \{\}$$

Delete_ SP_ 2

Se intenta eliminar un símbolo de la tabla, cuando:
- El símbolo a eliminar pertenece a los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- No hay ningún elemento en el conjunto formado por: el símbolo a eliminar.

Delete_SP_3

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

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

Delete_ SP_ 3

Se intenta eliminar un símbolo de la tabla, cuando:
- El símbolo a eliminar pertenece a los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- El símbolo a eliminar es el único elemento de los símbolos cargados en la tabla.

Delete_SP_4

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

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

Delete_ SP_ 4

Se intenta eliminar un símbolo de la tabla, cuando:
- El símbolo a eliminar pertenece los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por: el símbolo a eliminar.
- El símbolo a eliminar pertenece los símbolos cargados en la tabla.

Delete_SP_5

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

$s? \in \mathrm{dom}\ st$
$st \neq \{\}$
$\{s?\} \neq \{\}$
$\{s?\} \cap \mathrm{dom}\ st = \{\}$

Delete_ SP_ 5

Se intenta eliminar un símbolo de la tabla, cuando:
- El símbolo a eliminar pertenece los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por: el símbolo a eliminar.
- El conjunto formado por el símbolo a eliminar y los símbolos cargados en la tabla no tienen ningún elemento en común.

Delete_SP_7

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

$s? \in \mathrm{dom}\ st$
$st \neq \{\}$
$\{s?\} \cap \mathrm{dom}\ st \neq \{\}$
$\neg\ \mathrm{dom}\ st \subseteq \{s?\}$
$\neg\ \{s?\} \subseteq \mathrm{dom}\ st$

Delete_ SP_ 7

Se intenta eliminar un símbolo de la tabla, cuando:
- El símbolo a eliminar pertenece los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- El conjunto formado por el símbolo a eliminar y los símbolos cargados en la tabla tienen tienen al menos un elemento en común.
- Existe al menos un elemento en los símbolos cargados en la tabla que no pertenece al conjunto formado por: el símbolo a eliminar.
- El símbolo a eliminar no pertenece al los símbolos cargados en la tabla.

Delete_SP_12

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

$s? \notin \text{dom } st$
$st \neq \{\}$
$\{s?\} \neq \{\}$
$\{s?\} \cap \text{dom } st = \{\}$

Delete_ SP_ 12

Se intenta eliminar un símbolo de la tabla, cuando:

- El símbolo a eliminar no pertenece los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- Existe al menos un elemento en el conjunto formado por: el símbolo a eliminar.
- El conjunto formado por el símbolo a eliminar y los símbolos cargados en la tabla no tienen ningún elemento en común.

Delete_SP_13

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

$s? \notin \text{dom } st$
$st \neq \{\}$
$\{s?\} \cap \text{dom } st \neq \{\}$
$\text{dom } st \subset \{s?\}$

Delete_ SP_ 13

Se intenta eliminar un símbolo de la tabla, cuando:

- El símbolo a eliminar no pertenece los símbolos cargados en la tabla.
- Existe al menos un elemento en la tabla de símbolos.
- El conjunto formado por el símbolo a eliminar y los símbolos cargados en la tabla tienen tienen al menos un elemento en común.
- Los símbolos cargados en la tabla están incluidos en el conjunto formado por: el símbolo a eliminar.

Ejemplo: Banco

Especificación y designaciones

$[NCTA]$

$SALDO == \mathbb{N}$

$MENSAJES ::= ok \mid numeroClienteEnUso \mid noPoseeSaldoSuficiente \mid saldoNoNulo$

- Cajas de ahorro existentes en el banco $\approx cajas$
- Números de cuenta cargados en el banco $\approx dom\ cajas$
- Dinero depositado en la caja de ahorro de $x \approx caja\ x$

Banco

$$cajas : NCTA \pfun SALDO$$

- Número de cuenta del cliente $\approx num$

DepositarOk

$$\Delta Banco$$
$$num? : NCTA;\ m? : \mathbb{Z}$$

$$num? \in \mathrm{dom}\ cajas$$
$$m? > 0$$
$$cajas' = cajas \oplus \{num? \mapsto cajas\ num? + m?\}$$

$DepositarE1 == [\Xi Banco;\ num? : NCTA \mid num? \notin \mathrm{dom}\ cajas]$

$DepositarE2 == [\Xi Banco;\ m? : \mathbb{Z} \mid m? \leq 0]$

- Intenta depositar dinero en una cuenta $\approx Depositar$

$Depositar == DepositarOk \lor DepositarE1 \lor DepositarE2$

- Número de cuenta del nuevo cliente $\approx num$

NuevoClienteOk

$\Delta Banco$
$num? : NCTA$
$rep! : MENSAJES$

$num? \notin \text{dom } cajas$
$cajas' = cajas \cup \{num? \mapsto 0\}$
$rep! = ok$

NuevoClienteE

$\Xi Banco$
$num? : NCTA$
$rep! : MENSAJES$

$num? \in \text{dom } cajas$
$rep! = numeroClienteEnUso$

- Intenta cargar un nuevo cliente $\approx$ *NuevoCliente*

$NuevoCliente == NuevoClienteOk \lor NuevoClienteE$

- Número de cuenta del cliente $\approx$ *num*

ExtraerOk

$\Delta Banco$
$num? : NCTA$
$m? : SALDO$
$rep! : MENSAJES$

$num? \in \text{dom } cajas$
$0 < m?$
$m? \leq cajas\ num?$
$cajas' = cajas \oplus \{num? \mapsto (cajas\ num?) - m?\}$
$rep! = ok$

$ExtraerE1 == DepositarE1$

$ExtraerE2 == DepositarE2$

*ExtraerE*3

$\Xi Banco$
$num? : NCTA$
$m? : SALDO$
$rep! : MENSAJES$

$m? > cajas\ num?$
$num? \in \text{dom}\ cajas$
$rep! = noPoseeSaldoSuficiente$

- Intenta realizar una extracción de dinero ≈ *Extraer*

$ExtraerE == ExtraerE1 \lor ExtraerE2 \lor ExtraerE3$
$Extraer == ExtraerOk \lor ExtraerE$

- Número de cuenta del cliente ≈ *num*

PedirSaldoOk

$\Xi Banco$
$num? : NCTA$
$saldo! : SALDO$
$rep! : MENSAJES$

$num? \in \text{dom}\ cajas$
$saldo! = cajas\ num?$
$rep! = ok$

- Intenta consultar el saldo de un cliente ≈ *PedirSaldo*

$PedirSaldoE == DepositarE1$
$PedirSaldo == PedirSaldoOk \lor PedirSaldoE$

Clases de prueba y descripciones

NuevoCliente_SP_2

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \notin \text{dom}\ cajas$
$cajas = \{\}$
$\{num? \mapsto 0\} \neq \{\}$

NuevoCliente_SP_2

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- No hay ningún elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado por el par: número de cuenta y 0.

NuevoCliente_SP_4

$$cajas : NCTA \pfun SALDO$$
$$num? : NCTA$$

$$num? \notin \text{dom } cajas$$
$$cajas \neq \{\}$$
$$\{num? \mapsto 0\} \neq \{\}$$
$$\text{dom } cajas = \text{dom}\{num? \mapsto 0\}$$

NuevoCliente_SP_4

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado pope el par: número de cuenta y 0.
- El número de cuenta del nuevo cliente es el único elemento de los números de cuenta cargados en el banco.

NuevoCliente_SP_5

$$cajas : NCTA \pfun SALDO$$
$$num? : NCTA$$

$$num? \notin \text{dom } cajas$$
$$cajas \neq \{\}$$
$$\{num? \mapsto 0\} \neq \{\}$$
$$\text{dom}\{num? \mapsto 0\} \subset \text{dom } cajas$$

NuevoCliente_SP_5

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado pone el par: número de cuenta y 0.
- El número de cuenta del nuevo cliente pertenece a los números de cuenta cargados en el banco.

NuevoCliente_SP_6

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \notin \text{dom } cajas$
$cajas \neq \{\}$
$\{num? \mapsto 0\} \neq \{\}$
$(\text{dom } cajas \cap \text{dom}\{num? \mapsto 0\}) = \{\}$

NuevoCliente_SP_6

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado pode el par: número de cuenta y 0.
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.

NuevoCliente_SP_7

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \notin \text{dom } cajas$
$cajas \neq \{\}$
$\{num? \mapsto 0\} \neq \{\}$
$\text{dom } cajas \subset \text{dom}\{num? \mapsto 0\}$

NuevoCliente_SP_7

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Los números de cuenta cargados en el banco están incluidos en el conjunto formado por el número de cuenta del nuevo cliente.

NuevoCliente_SP_8

$$
\begin{array}{l}
cajas : NCTA \pfun SALDO \\
num? : NCTA \\
\hline
num? \notin \dom cajas \\
cajas \neq \{\} \\
\{num? \mapsto 0\} \neq \{\} \\
(\dom cajas \cap \dom\{num? \mapsto 0\}) \neq \{\} \\
\neg\ \dom\{num? \mapsto 0\} \subseteq \dom cajas \\
\neg\ \dom cajas \subseteq \dom\{num? \mapsto 0\}
\end{array}
$$

NuevoCliente_SP_8

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado por el par: número de cuenta y 0.
- El número de cuenta del nuevo cliente no pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en los números de cuenta cargados en el banco que no está en el conjunto formado por el número de cuenta del nuevo cliente.

NuevoCliente_SP_12

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \in \text{dom } cajas$
$cajas \neq \{\}$
$\{num? \mapsto 0\} \neq \{\}$
$\text{dom } cajas = \text{dom}\{num? \mapsto 0\}$

NuevoCliente_SP_12

Se intenta cargar un nuevo cliente, cuando:
- El número de cuenta del nuevo cliente pertenece a los números de cuenta cargados en el banco.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Existe al menos un elemento en el conjunto formado por el par: número de cuenta y 0.
- El número de cuenta del nuevo cliente es el único elemento de los números de cuenta cargados en el banco.

PedirSaldo_SP_1

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \in \text{dom } cajas$
$\text{dom } cajas = \{num?\}$

PedirSaldo_SP_1

Se intenta consultar el saldo de un cliente, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El número de cuenta indicado es el único elemento de los números de cuenta cargados en el banco.

PedirSaldo_SP_2

$cajas : NCTA \pfun SALDO$
$num? : NCTA$

$num? \in \dom cajas$
$\dom cajas \neq \{num?\}$
$num? \in \dom cajas$

PedirSaldo_SP_2

Se intenta consultar el saldo de un cliente, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- Los números de cuenta cargados en el banco no son iguales al conjunto formado por el número de cuenta indicado.
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.

Depositar_SP_3

$cajas : NCTA \pfun SALDO$
$num? : NCTA$
$m? : \mathbb{Z}$

$num? \in \dom cajas$
$m? > 0$
$cajas \neq \{\}$

Depositar_SP_3

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.

Depositar_SP_4

$$cajas : NCTA \pfun SALDO$$
$$num? : NCTA$$
$$m? : \mathbb{Z}$$

$$num? \in \mathrm{dom}\ cajas$$
$$m? > 0$$
$$cajas \neq \{\}$$
$$\mathrm{dom}\ cajas = \mathrm{dom}\{num? \mapsto (cajas\ num? + m?)\}$$

Depositar_SP_4

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- El número de cuenta indicado es el único elemento de los números de cuenta cargados en el banco.

Depositar_SP_5

$$cajas : NCTA \pfun SALDO$$
$$num? : NCTA$$
$$m? : \mathbb{Z}$$

$$num? \in \mathrm{dom}\ cajas$$
$$m? > 0$$
$$cajas \neq \{\}$$
$$\mathrm{dom}\{num? \mapsto (cajas\ num? + m?)\} \subset \mathrm{dom}\ cajas$$

Depositar_SP_5

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.

Depositar_SP_6

$$
\begin{array}{l}
cajas : NCTA \pfun SALDO \\
num? : NCTA \\
m? : \mathbb{Z} \\
\hline
num? \in \dom cajas \\
m? > 0 \\
cajas \neq \{\} \\
(\dom cajas \cap \dom\{num? \mapsto (cajas\ num? + m?)\}) = \{\}
\end{array}
$$

Depositar_SP_6

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- El número de cuenta indicado no pertenece a los números de cuenta cargados en el banco.

Depositar_SP_7

$$
\begin{array}{l}
cajas : NCTA \pfun SALDO \\
num? : NCTA \\
m? : \mathbb{Z} \\
\hline
num? \in \dom cajas \\
m? > 0 \\
cajas \neq \{\} \\
\dom cajas \subset \dom\{num? \mapsto (cajas\ num? + m?)\}
\end{array}
$$

Depositar_SP_7

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- Los números de cuenta cargados en el banco están incluidos en el conjunto formado por el número de cuenta indicado.

$$
\begin{array}{|l}
\textit{Depositar\_SP\_8} \\
\hline
cajas : NCTA \pfun SALDO \\
num? : NCTA \\
m? : \mathbb{Z} \\
\hline
num? \in \mathrm{dom}\ cajas \\
m? > 0 \\
cajas \neq \{\} \\
(\mathrm{dom}\ cajas \cap \mathrm{dom}\{num? \mapsto (cajas\ num? + m?)\}) \neq \{\} \\
\neg\ \mathrm{dom}\{num? \mapsto (cajas\ num? + m?)\} \subseteq \mathrm{dom}\ cajas \\
\neg\ \mathrm{dom}\ cajas \subseteq \mathrm{dom}\{num? \mapsto (cajas\ num? + m?)\} \\
\hline
\end{array}
$$

Depositar_SP_8

Se intenta depositar dinero en una cuenta, cuando:
- El número de cuenta indicado pertenece a los números de cuenta cargados en el banco.
- El monto a depositar es positivo.
- Existe al menos un elemento en las cajas de ahorro existentes en el banco.
- El número de cuenta indicado no pertenece a los números de cuenta cargados en el banco
- Existe al menos un elemento en los números de cuenta cargados en el banco que no está en el conjunto formado por el número de cuenta indicado.

Ejemplo: Sistema de sensores

Especificación y designaciones

- x es un identificador de sensor válido $\approx x \in SENSOR$

B

$[SENSOR]$

- Conjunto de identificadores válidos $\approx dom\ smax$
- x es un identificador válido $\approx x \in dom\ smax$
- Valor máximo registrado para $x \approx smax\ x$

$MaxReadings == [smax : SENSOR \pfun \mathbb{Z}]$

- Identificador del sensor leído $\approx s?$
- Valor de medición leído $\approx r?$

$$
\begin{array}{|l}
\hline
\textit{KeepMaxReadingOk} \\
\Delta \textit{MaxReadings} \\
s? : \textit{SENSOR};\ r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \textit{smax} \\
\textit{smax}\ s? < r? \\
\textit{smax}' = \textit{smax} \oplus \{s? \mapsto r?\} \\
\hline
\end{array}
$$

$$\textit{KeepMaxReadingE1} == [\Xi \textit{MaxReadings};\ s? : \textit{SENSOR} \mid s? \notin \operatorname{dom} \textit{smax}]$$

$$
\begin{array}{|l}
\hline
\textit{KeepMaxReadingE2} \\
\Xi \textit{MaxReadings} \\
s? : \textit{SENSOR};\ r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \textit{smax} \\
r? \leq \textit{smax}\ s? \\
\hline
\end{array}
$$

- Intenta actualizar un valor máximo sensado $\approx \textit{KeepMaxReading}$

$$\textit{KeepMaxReading} == \textit{KeepMaxReadingOk} \lor \textit{KeepMaxReadingE1} \lor \textit{KeepMaxReadingE2}$$

Clases de prueba y descripciones

$$
\begin{array}{|l}
\hline
\textit{KeepMaxReading\_SP\_3} \\
\textit{smax} : \textit{SENSOR} \pfun \mathbb{Z} \\
s? : \textit{SENSOR} \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \textit{smax} \\
\textit{smax}\ s? < r? \\
\textit{smax}\ s? < 0 \\
r? > 0 \\
\hline
\end{array}
$$

KeepMaxReading_SP_3

Se intenta actualizar un valor máximo sensado, cuando:
- El identificador del sensor leído pertenece al conjunto de identificadores válidos.
- El valor máximo registrado para el identificador del sensor leído es menor a el valor de medición leído.
- El valor máximo registrado para el identificador del sensor leído es negativo.
- El valor de medición leído es positivo.

KeepMaxReading_SP_4

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \operatorname{dom} smax$$
$$smax\ s? < r?$$
$$smax\ s? = 0$$
$$r? > 0$$

KeepMaxReading_SP_4

Se intenta actualizar un valor máximo sensado, cuando:
- El identificador del sensor leído pertenece al conjunto de identificadores válidos.
- El valor máximo registrado para el identificador del sensor leído es menor a el valor de medición leído.
- El valor máximo registrado para el identificador del sensor leído es igual a cero.
- El valor de medición leído es positivo.

KeepMaxReading_SP_7

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \notin \operatorname{dom} smax$$
$$smax\ s? < 0$$
$$r? = 0$$

KeepMaxReading_SP_7

Se intenta actualizar un valor máximo sensado, cuando:
- El identificador del sensor leído no pertenece al conjunto de identificadores válidos.
- El valor máximo registrado para el identificador del sensor leído es negativo.
- El valor de medición leído es igual a cero.

KeepMaxReading_SP_14

$$
\begin{array}{l}
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} smax \\
r? \leq smax\ s? \\
smax\ s? = 0 \\
r? > 0
\end{array}
$$

KeepMaxReading_SP_14

Se intenta actualizar un valor máximo sensado, cuando:
- El identificador del sensor leído pertenece al conjunto de identificadores válidos.
- El valor de medición leído es menor o igual a el valor máximo registrado para el identificador del sensor leído.
- El valor máximo registrado para el identificador del sensor leído es positivo.
- El valor de medición leído es positivo.

Apéndice B

Guía de estilo para designaciones

Las designaciones son la principal fuente de conocimiento del dominio. Éstas son fundamentales para que nuestro sistema de NLG pueda generar descripciones independientes del dominio de aplicación. A continuación, enumeraremos una serie de pautas que el ingeniero realizando la especificación debe tener en cuenta para que nuestro sistema produzca textos lo más fluidos y naturales posibles.

A fin de un correcto funcionamiento de nuestro sistema, resulta indispensable designar los siguientes elementos de una especificación:

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

Cuando el objeto a designar sea una función es recomendable designar:

1. La función (f)
2. El dominio de la función (dom f)
3. La aplicación de la función (f x)
4. El rango de la función (ran f)

Esto se debe a que es muy común que aparezcan en las clases de prueba generadas por el TTF el dominio como la aplicación de función y observamos que es posible obtener mejores resultados si se encuentran designadas las tres expresiones enumeradas anteriormente. De lo contrario nuestro sistema describirá los casos anteriores utilizando la terminología de funciones (dominio, aplicación, etc.) y puede hacer la interpretación del texto en término del dominio más compleja.

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.

[EdA14] Real Academia Española and E.A. de Academias. *Diccionario de la lengua española*. Planeta Publishing Corporation, 2014.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[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.