Capítulo 1

Introducción

El testing basado en modelos es una de las técnicas de testing más prometedoras para la verificación de software crítico. Estas metodologías comienzan con un modelo formal o especificación del software, de la cual son generados los casos de prueba.

Un caso particular del testing basado en modelos es el *Test Template Framework* (TTF), descrito por Stocks y Carrington [**?**], el cual utiliza, como modelo de entrada, especificaciones formales escritas en notación Z y establece como generar casos de prueba para cada operación incluida en el modelo. Esta técnica genera descripciones lógicas, también en lenguaje Z, de los casos de prueba.

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, en particular, si no son capaces de leer notación Z, no podrán entender que está siendo testeado. En estos casos, una descripción en lenguaje natural de cada caso de prueba debería acompañar a los mismos a fin de hacerlos accesibles para los expertos en el dominio.

El objetivo de este trabajo, será entonces, generar descripciones en lenguaje natural para los casos de prueba generados por el TTF. Anteriormente Cristiá y Plüss [**?**] desarrollaron una solución ad-hoc basada en templates (dependiente del dominio de aplicación y cantidad de operaciones) para generar descripciones los casos de prueba de un software para un satélite.

Basado en el trabajo antes mencionado desarrollaremos una solución independiente del dominio de aplicación y del número de operaciones del sistema, principalmente utilizando información contenida en las designaciones [**?**] y trabajando fundamentalmente con las clases de prueba, las que nos permiten generar mejores descripciones.

En particular trabajaremos con Fastest[1], una implementación del TTF desarrollada por Crstiá y Monetti [**?**] capaz de generar casos de prueba a

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

partir de una especificación Z. Además, el sistema de NLG desarrollado en este trabajo fue implementado Java e integrado a la implementación de Fastest permitiendo generar descripciones de los casos de prueba interactivamente desde la herramienta.

TODO acá podría ir una breve descripción de cada uno de los capítulos siguientes.

Capítulo 2

Test Template Framework

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

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

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

2.1. Ejemplo: Symbol Table.

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

$[SYM, VAL]$

$REPORT ::= ok \mid symbolNotPresent$

ST

$st : SYM \pfun VAL$

Update

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

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

LookUpOk

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

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

LookUpE

ΞST
$s? : SYM$
$rep! : REPORT$

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

$LookUp == LookUpOk \lor LookUpE$

Figura 2.1: Modelo Z para una tabla de símbolos.

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

$$[SYM, VAL]$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

LookUp_DNF_1

LookUp_VIS

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

LookUp_SP_1

LookUp_DNF_1

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

LookUp_SP_2

LookUp_DNF_1

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

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

LookUp_DNF_2

LookUp_VIS

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

LookUp_SP_4

LookUp_DNF_2

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

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

Figura 2.2: Clases de prueba generadas para operación LookUp.

2.3. Fastest.

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

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

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

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

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

Capítulo 3

Designaciones

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

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

termino_formal $\approx$ *texto en lenguaje natural*

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

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

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

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

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

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

3.1. Que designar en una especificación Z

TODO! esta bien ubicado esto acá ?

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

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

Siguiendo lo propuesto por Jackson, si en la especificación apareciera el término: ActuaronJuntosPelicula(x, y) -verdadero para dos actores que hayan actuado juntos en alguna película-, éste podría definirse en términos las designaciones anteriores y no debería designarse. Pero en este caso para generar, por ejemplo, el texto "Al Pacino y Marlon Brando actuaron juntos en la película El Padrino" a partir de *ActuoEn("Al Pacino", "El Padrino")* y *ActuoEn("Marlon Brando", "El Padrino")* necesitaríamos procesar las designaciones teniendo en cuenta cuestiones referentes al dominio de aplicación. Y esto atentaría contra nuestros intereses de lograr un sistema de generación de lenguaje natural independiente del dominio de aplicación.

Por otro lado creemos que también resultará de utilidad y otorgará al sistema una mayor flexibilidad permitir al usuario escribir designaciones que puedan resultar "redundantes" para la comprensión de la especificación pero

de gran ayuda a la hora de generar una descripción en particular. En estos casos cuando nuestro sistema deba describir uno de estos términos designados hará uso del texto incluido en la designación en lugar de intentar describir el mismo como se verá mas adelante.

Finalmente, a partir de un análisis de textos generados para distintas especificaciones, creemos que para tener descripciones fluidas es recomendable designar:

(TODO: sacados del trabajo de federico moya, falta completar a partir de análisis de textos generados)

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

Capítulo 4

Generación de lenguaje natural.

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

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

4.1. Análisis de requerimientos

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

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

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

4.2. Tareas de la generación de lenguaje natural

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

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

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

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

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

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

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

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

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

4.3. Arquitectura para la NLG.

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

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

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

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

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

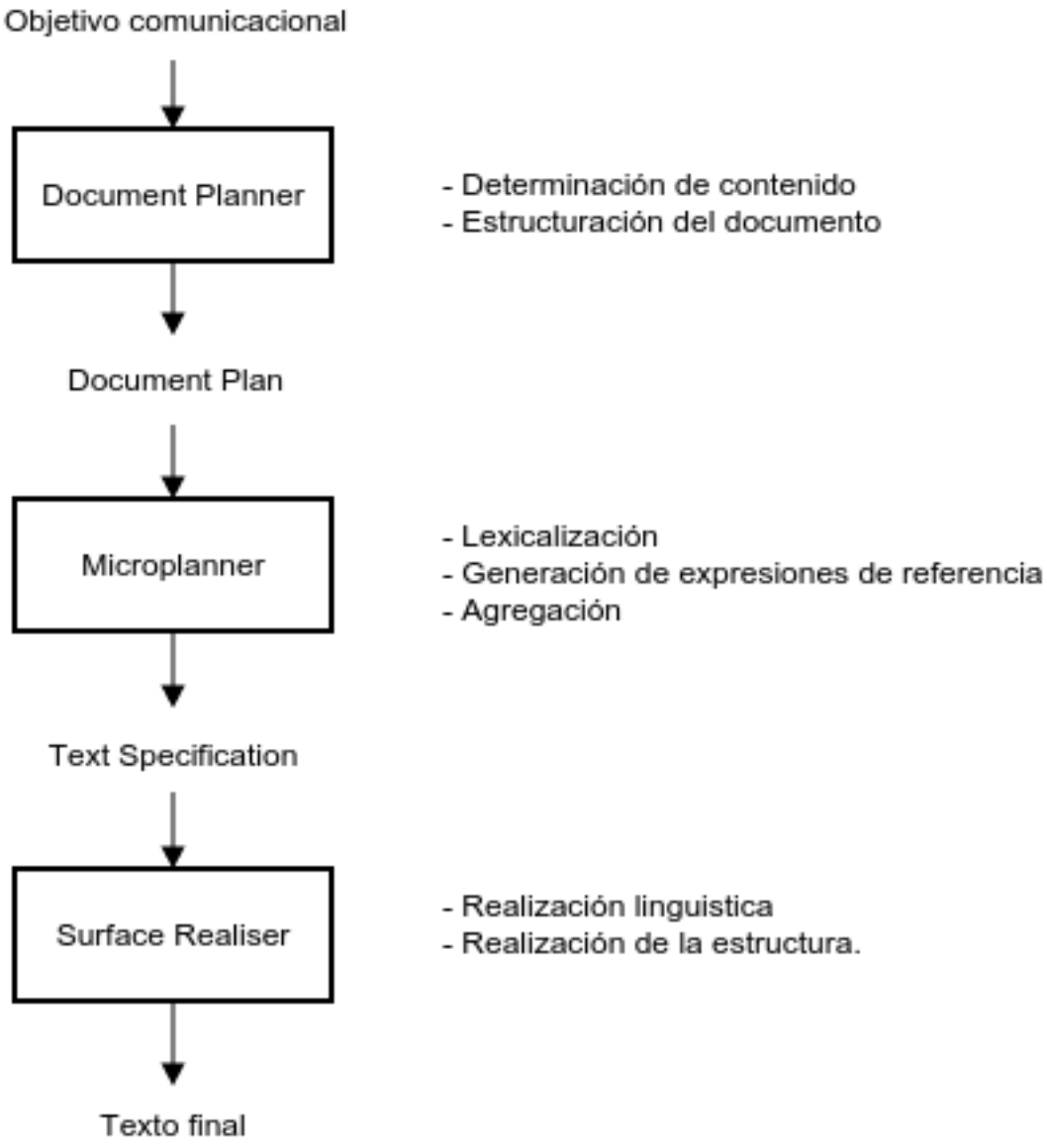

Figura 4.1: Arquitectura típica sistema NLG.

Capítulo 5

Descripciones

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

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

5.1. Corpus de descripciones

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

Para nuestro trabajo debemos recolectar un conjunto lo suficientemente amplio de ejemplos de clases de prueba que caractericen la variedad de textos que deseamos generar, luego una persona capacitada de leer lenguaje Z deberá describir en lenguaje natural las mismos. Finalmente se deberá revisar este *corpus* inicial y modificarlo si existe algún caso en el que haya alguna descripción que resulte técnicamente imposible de generar o prohibitivamente cara a nivel computacional. Una vez finalizado este proceso tendremos en nuestro poder un *corpus objetivo* que nos servirá para sustentar muchas de las decisiones que tomemos a lo largo del trabajo. Este también nos servirá para realizar una evaluación del sistema una vez implementado, comparando

los textos generados por el nuestro sistema con las descripciones del *corpus* realizadas por una persona.

En el Apéndice **??** se encuentra el corpus utilizado para este trabajo. Para elaborar el mismo recolectamos una serie de clases y casos de prueba generados con Fastest a partir de distintas especificaciones y luego se escribimos manualmente cada una de las descripciones estas clases de prueba. Con las especificaciones y clases de prueba incluidas en el *corpus*, intentamos abarcar todo el rango de textos que esperamos que nuestro sistema sea capaz de producir, para esto tuvimos en cuenta incluir clases de pruebas que cubran todas las expresiones de Z contempladas dentro del alcance de este trabajo y sus posibles combinaciones. También trabajamos con especificaciones sobre distintos dominios de aplicación con el fin de lograr *corpus* que nos sea de utilidad para dar con una solución que sea independiente del dominio de aplicación.

La Figura 5.2 muestra a modo de ejemplo una descripción para la clase de prueba *LookUp_SP_1* generada a partir de la especificación para una tabla de símbolos introducida anteriormente (pág. 4). El *corpus de descripciones* utilizado para este trabajo, constará entonces de una colección de ejemplos como el anterior.

LookUp_SP_1

$$
\begin{array}{|l}
\hline
\textit{LookUp\_VIS} \\
\hline
s? \in \operatorname{dom} st \\
\operatorname{dom} st = \{s?\} \\
\hline
\end{array}
$$

Figura 5.1: Clase de prueba para operación LookUp.

Descripciones SymbolTable

LookUp_SP_1: Se busca un símbolo en la tabla.

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

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

Capítulo 6

Document Planning

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

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

6.1. Entrada y salida del document planner

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

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

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

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

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

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

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

6.2. Representación del dominio

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

El *corpus de descripciones* (apéndice **??**) resulta una buena fuente para estudiar y definir como deberemos representar la información o *mensajes*[1] a comunicar. Podemos ver en todos los ejemplos del *corpus* una relación directa entre la información a comunicar y las expresiones Z pertenecientes al dominio de aplicación, donde podemos observar que cada linea de una descripción se corresponde perfectamente con la verbalización de la expresión en lenguaje Z correspondiente. Veamos por ejemplo las siguientes líneas de la descripción de LookUp_SP_1 introducida anteriormente (pág. 18):

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

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

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

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

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

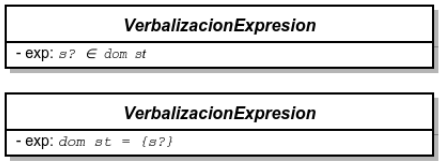

Figura 6.1: Mensajes a comunicar para el ejemplo de la figura 5.2.

6.3. Determinación del contenido

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

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

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

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

Update_SP_4

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

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

Figura 6.2: Clase de prueba para operación Update (pág. 4).

Podemos observar que la siguiente expresión del ejemplo anterior:

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

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

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

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

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

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

podríamos describirla como[3]:

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

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

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

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

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

que luego podríamos describir como:

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

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

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

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

6.4. Estructuración del documento

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

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

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

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

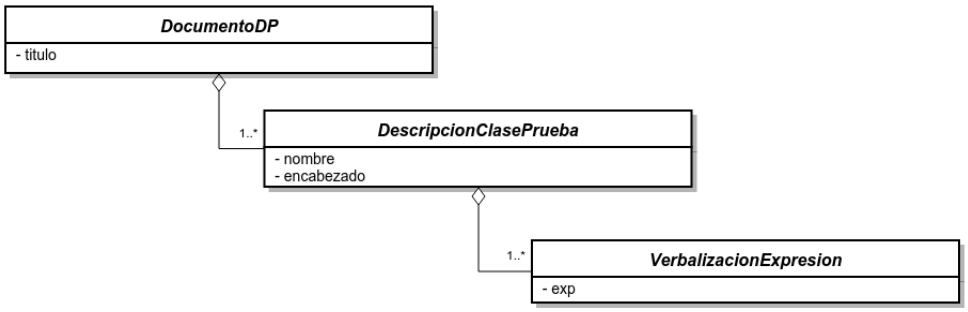

Figura 6.3: Document plan.

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

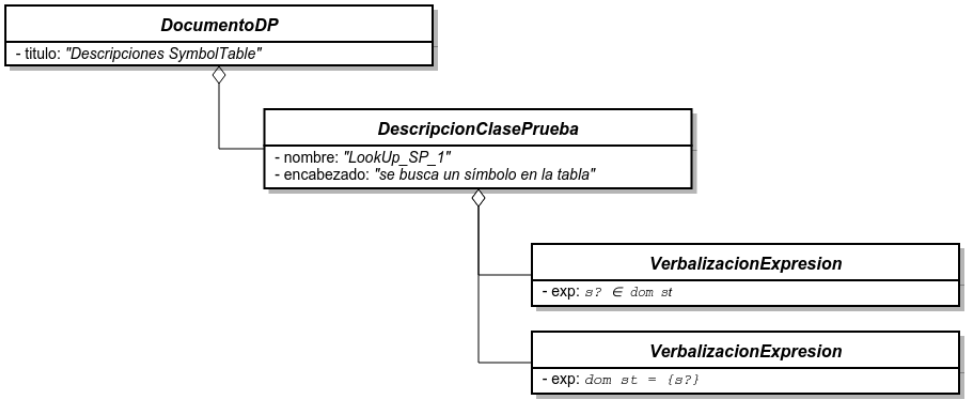

Figura 6.4: Document plan correspondiente al texto de la figura 5.1.

Capítulo 7

Microplanning.

La tarea del microplanner será tomar el document plan generado en la etapa anterior y refinarlo a modo de producir una especificación mas detallada del texto.

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

Lexicalización. Elegir que palabras particulares, constructores sintácticos y anotaciones de "mark-up" usar para comunicar la información contenida en el document plan.

Agregación. Decidir cuanta información debe ser comunicada en cada oración del texto final.

Generación de expresiones de referencia. Determinar que frases deben ser usadas para identificar las entidades particulares del dominio de aplicación.

Cabe aclarar que el resultado de esta etapa no será todavía un texto, sino que quedarán por tomar decisiones acerca de la sintaxis, morfología y cuestiones de presentación, de las cuales se encargará la siguiente etapa.

Tanto la tarea de lexicalización como la de generación de expresiones de referencia se encargan de "mapear" elementos del dominio de aplicación a elementos lingüísticos, y pueden parecer bastante similares. Sin embargo, la diferencia radica en que la lexicalización se encargará de expresar lingüísticamente las relaciones y conceptos de nuestro dominio de aplicación, mientras que la tarea de generación de expresiones de referencia se encargará de identificar las entidades de nuestro dominio. Se dividen de esta manera porque los algoritmos requeridos para cada tarea son diferentes en su naturaleza.

A continuación, estudiaremos la entrada y salida del microplanner, luego veremos la arquitectura utilizada y describiremos mas en profundidad las tareas de lexicalización y generación de expresiones de referencia realizadas en este trabajo. Creemos que si bien es posible realizar tareas de agregación, no se encuentra dentro del alcance de este trabajo, sin embargo veremos algunas ideas respecto a este último punto en el capítulo **??**.

7.1. Entrada y salida del microplanner

7.2. Arquitectura

7.3. Lexicalización

7.4. Generación de expresiones de referencia