Definición de estrategias para la aplicación automática de tácticas de testing en el marco del TTF y Fastest

Joaquín Cuenca

Director: Dr. Maximiliano Cristiá
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

27 de agosto de 2014

Resumen

En el testing basado en modelos (MBT), se busca generar casos de prueba para testear un programa utilizando únicamente un modelo del sistema en cuestión. El Test Template Framework (TTF) es un método de MBT que permite obtener casos de prueba abstractos a partir de una especificación Z del sistema. Para esto, define tácticas de testing que se aplican sobre el espacio de entrada de una operación de la especificación, particionándolo de distintas maneras según como sean aplicadas. Esto genera clases de prueba de las cuales se derivan los casos de prueba abstractos a utilizar en el testing.

Fastest es una herramienta que automatiza gran parte del proceso de testing descripto por el TTF, pero no realiza de forma automática la generación de las clases de prueba, es decir, es el usuario quien debe aplicar manual y sucesivamente las tácticas a fin de dar forma y orientar el testing a las áreas deseadas. Este proceso requiere no sólo de un esfuerzo considerable por parte del usuario, sino también un gran conocimiento de los detalles de la especificación. En este trabajo se definen estrategias de testing las cuales combinan y aplican de forma automática las tácticas definidas en el TTF y Fastest con el fin de automatizar en mayor medida el proceso de generación de casos de prueba. Además, se define un lenguaje para la definición de estrategias de testing en Fastest.

Parte de esta tesina fue publicada en el siguiente artículo:

Maximiliano Cristiá, Joaquín Cuenca and Claudia Frydman: Coverage Criteria for Logical Specifications. In 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST 2014) affiliated to the Brazilian Conference on Software: Theory and Practice (CBSoft).

Índice general

Agradecimientos

A mis viejos y mi hermano, por estar siempre, por aconsejarme, por guiarme, por el apoyo incondicional. Hoy no sería quien soy sin ustedes.

A Martu, mi novia, por compartir y acompañarme en todas. A tu lado disfruté plenamente este camino.

A mis amigos, Euge, Mau, Toto y Joaco, quienes me han enseñado y nos hemos divertido tanto en esta travesía, que me hacen dudar de querer llegar al final.

A Maxi, por el apoyo y la paciencia constante durante toda la realización de este trabajo, nada de esto lo podría haber hecho sin tu ayuda.

Y a toda la LCC, por ser tan cálida, tan familiar, todo fue más fácil en este ambiente.

Muchas gracias a todos!

Capítulo 1

Introducción

A lo largo del tiempo, el testing ha demostrado su importancia como parte del proceso de desarrollo de software. Muchos autores han marcado las limitaciones del testing, bajo las cuales sólo es posible encontrar errores en el programa, y no demostrar su corrección. Pero más allá de esta limitación, el testing ha ganado su lugar en la industria del software como la técnica de verificación más utilizada. Además ha demostrado que puede ser realmente efectivo si se lo practica de forma rigurosa y ordenada.

Cuando se empezó a conformar una comunidad del software, la cantidad de personas abocadas a cada proyecto de desarrollo no era mucha y los sistemas no contaban con los niveles de complejidad que se observan hoy en día, por lo que muchas veces era el mismo programador quien desarrollaba el código y quien lo controlaba.

A medida que la complejidad de los sistemas fue en aumento, también lo hizo la cantidad de personas involucradas en los proyectos. Este crecimiento marcó la necesidad de técnicas que otorgaran mayor seguridad sobre si los productos desarrollados se comportaban de la manera pensada. En 1979, Glenford J. Myers [1] introduce el concepto de testing de software como se lo conoce hoy en día, además de otras cuestiones relacionadas. Esto generó una separación en las técnicas de control utilizadas, la cual logró encasillar y aislar las actividades propias del desarrollo de aquellas propias de la verificación.

Con el paso del tiempo, se comenzaron a utilizar métodos formales pa-

ra especificar y diseñar. Si bien fueron introducidos con el objetivo de lograr pruebas de corrección (entre otras cosas), las especificaciones formales comenzaron a destacarse dentro del testing de software. Esto se debió al alcance reducido que lograban tener por ese momento las especificaciones informales, las cuales si bien demostraron ser necesarias, tienen una utilidad limitada. Es en este período donde surge la idea de la imposibilidad de hacer testing sin contar con una especificación, cualquiera sea su tipo.

El testing basado en modelos (MBT) es una colección de métodos orientados a la generación de casos de testing a partir del análisis de un modelo del sistema que se desea testear [2]. Los métodos de MBT han conseguido muy buenos resultados tanto en la teoría como en la práctica en los últimos años [3, 4, 5, 6, 7].

Phil Stocks y David Carrington definieron el Test Template Framework (TTF) [9], un método de MBT que genera casos de prueba a partir de especificaciones Z [11]. El TTF brinda una serie de mecanismos que permiten analizar la especificación para obtener casos de prueba con los que realizar el testing de la implementación. Si bien las publicaciones originalmente utilizaron el lenguaje formal de especificaciones Z, fue pensado para trabajar de forma independiente al lenguaje.

Desde hace ya varios, años la comunidad de testing trabaja en la automatización del proceso de testing como una medida para reducir sus costos y mejorar su eficiencia. Fastest, es una herramienta que contribuye a este asunto al implementar la teoría del TTF [13]. Actualmente permite generar casos de prueba de forma casi totalmente automática partiendo de una especificación formal Z.

Este trabajo intenta automatizar en mayor medida una de las etapas del proceso de testing definido en el TTF e implementado más tarde en Fastest. La herramienta permite la aplicación de las tácticas de testing, de forma tal que el usuario debe indicar qué tácticas aplicar y con qué argumentos, y estas son computadas. Sin embargo, la determinación de estos valores y la elección de las tácticas a usar requiere de quien realiza la tarea que posea conocimientos tanto del lenguaje Z como del TTF y sus tácticas de testing, además del tiempo que conlleva realizar esta tarea manualmente. Entonces

proponemos la unión de tácticas de testing en lo que llamamos estrategias de testing; las cuales aplican las tácticas, con las que fueron definidas, de forma completamente automática. El propósito de las estrategias no es solo proveer un nuevo nivel de automatización, sino, fundamentalmente, darle a los usuarios una forma más abstracta y orientada al testing para la generación de casos de prueba. En parte logramos esto mediante la organización de las estrategias de testing de acuerdo a un orden parcial. Por otro lado, creamos un lenguaje para definir estrategias en Fastest.

El concepto de estrategia de testing requiere primero que se introduzca el TTF haciendo énfasis en el trabajo manual que los usuarios de Fastest necesitan realizar para aplicar las tácticas de testing (Capítulo 2). También requiere de la descripción de las tácticas disponibles (Capítulo 3). Con estos elementos, las estrategias de testing son introducidas en el Capítulo 4 y se presenta un lenguaje para definirlas en el Capítulo 5. Finalmente, damos una pequeña conclusión en el Capítulo 6.

Capítulo 2

Sobre la notación Z, el Test Template Framework y Fastest

En este capítulo seguiremos un ejemplo que nos permitirá introducir los conceptos necesarios para comprender el trabajo que debe realizar el usuario de Fastest para generar casos de prueba. De esta forma podremos explicar nuestro aporte en los capítulos posteriores. En particular, queremos enfocarnos en el proceso de aplicación de tácticas de testing, pero para eso debemos introducir primero la notación Z y el TTF.

2.1. La notación Z

Z es un lenguaje formal de especificaciones basado en la lógica de primer orden, el cálculo lambda y la teoría de conjuntos de Zermelo-Fraenkel, que ha sido utilizado en una gran cantidad de sistemas de software. Actualmente hay dos versiones del lenguaje, la primera fue descripta por Spivey [11], la cual fue más tarde estandarizada por los procesos de las normas ISO dando lugar a la versión que se conoce como estándar [12]. Como Fastest utiliza esta última, nos enfocaremos en ella en el siguiente ejemplo introductorio.

2.1.1. Especificando en Z

En su presentación original, Stocks y Carrington introdujeron el TTF utilizando como ejemplo la especificación formal de una tabla de símbolos [9]. Nosotros utilizaremos como ejemplo una pequeña lista de reservas, como parte del sistema de administración de un negocio, la cual nos permitirá mostrar en detalle el tipo de trabajo manual que debe realizar el usuario.

Especificaremos un pequeño sistema para almacenar información sobre los productos disponibles en un negocio, y operaciones para que un cliente pueda reservar alguno de ellos. Debemos administrar una tabla de reservas, donde se almacenan referencias entre productos y algunos datos correspondientes a los mismos, como ser, su estado (de reserva), su precio y la información del cliente que lo ha reservado. Además, especificaremos operaciones para modificar estos datos. Una especificación más completa, propondría operaciones tanto para modificar la información almacenada, como también operaciones para consultarla, pero en este caso nos centraremos únicamente en una operación de modificación, ya que es suficiente para introducir los conceptos que necesitaremos más adelante.

Para empezar, necesitamos una manera de identificar cada producto. Como los utilizaremos únicamente como identificadores para obtener ciertos datos, los podemos abstraer, sin preocuparnos por su estructura interna. De la misma manera, definiremos los identificadores que utilizaremos para representar a los clientes. Para estos casos, Z provee los llamados tipos básicos, donde la sintaxis para definirlos es la siguiente:

$$[PRODUCTO]$$
$$[CLIENTE]$$

De esta manera, es posible declarar una variable de tipo *PRODUCTO* o *CLIENTE* y es posible también definir tipos más complejos a partir de ellos. Por otro lado, como deseamos almacenar información sobre el costo de cada uno de los productos, definiremos el tipo donde se alojarán estos valores. Para ello utilizaremos los números enteros:

$$VALOR == \mathbb{Z}$$

Entonces, podemos definir nuestra tabla, o mejor dicho, una estructura que concentrará toda la información del sistema, de la siguiente manera:

Reservas

$$precio : PRODUCTO \pfun VALOR$$
$$contacto : PRODUCTO \pfun CLIENTE$$

Este tipo de definiciones se denominan esquemas. Cada esquema tiene un nombre, como *Reservas*, los cuales pueden ser utilizados en las construcciones de otros esquemas. En este caso en particular, el esquema es de estado, ya que solo define variables de estado. En el ejemplo son *precio* y *contacto*. De esta manera, cada estado del sistema corresponde a un valor para estas variables. El constructor $\pfun$ define funciones parciales, entonces, *precio* es una función parcial, tal que recibe como argumento un elemento de *PRODUCTO* y devuelve un elemento de *VALOR*, y de la misma manera *contacto* es una función parcial de *PRODUCTO* a *CLIENTE*.

Podemos ahora definir el estado inicial del sistema:

InitReservas

$$Reservas$$

$$precio = \emptyset$$
$$contacto = \emptyset$$

InitReservas es un nuevo esquema, el cual definimos en dos partes. La parte superior es la declaración y la inferior el predicado. En la sección de declaración podemos tanto definir variables como incluir esquemas. En este caso incluimos el esquema *Reservas* definido previamente. De esta manera, las variables definidas en este último, se encuentran presentes en el nuevo esquema. El predicado indica que las variables son iguales al conjunto vacío. El símbolo = hace referencia a la igualdad lógica y no al concepto de asig-

nación presente en los lenguajes imperativos, ya que en Z no se manejan los conceptos de control de flujo. En Z, las funciones son conjuntos de pares ordenados, por lo tanto pueden ser comparadas con el conjunto vacío.

Definimos, también, el tipo de los mensajes que reportarán las operaciones, los cuales usaremos para indicar si una operación determinada se pudo realizar o si se encontró un inconveniente. Las variables de este tipo, podrán tener sólo los valores, *ok*, *productoNoExistente* o *productoNoDisponible*:

$$MENSAJE ::= ok \mid productoNoExistente \mid productoNoDisponible$$

Ahora que tenemos las definiciones necesarias y el estado inicial del sistema, podemos crear las operaciones para modificarlo. De esta forma definimos *ReservarOk*, que especifica el comportamiento del sistema cuando un determinado cliente realiza correctamente la reserva de un producto:

ReservarOk

$$\Delta Reservas$$
$$p? : PRODUCTO$$
$$c? : CLIENTE$$
$$msg! : MENSAJE$$

$$p? \in \operatorname{dom} precio$$
$$p? \notin \operatorname{dom} contacto$$
$$contacto' = contacto \cup \{p? \mapsto c?\}$$
$$precio' = precio$$
$$msg! = ok$$

La primera expresión, $\Delta Reservas$, es una abreviación para incluir los esquemas *Reservas* y *Reservas*$'$. Como vimos anteriormente, incluir *Reservas* equivale a definir las variables *precio* y *contacto*, y de la misma manera, incluir *Reservas*$'$ es definir las variables de *Reservas*, pero decoradas con una comilla simple. De esta manera el esquema define también las variables *precio*$'$ y *contacto*$'$, que representan el estado de las variables posterior a la

aplicación de la operación. Entonces, utilizamos el símbolo Δ como sinónimo de que la operación realiza una modificación de estado.

Luego definimos las variables p y c, decoradas con el símbolo ?, lo que indica que son variables de entrada. En este caso, tienen tipo *PRODUCTO* y *CLIENTE* respectivamente, es decir, la entrada de la operación consta de un producto y un cliente, que indican quién reservará qué producto. Y la variable *msg*, indicada con !, representa el mensaje de salida emitido por la operación.

El predicado consta de cinco predicados atómicos. En Z, cuando los predicados están escritos en líneas separadas, se asume que están conjugados, es decir que el predicado:

$$
\begin{aligned}
&p? \in \operatorname{dom} precio \\
&p? \notin \operatorname{dom} contacto \\
&contacto' = contacto \cup \{p? \mapsto c?\} \\
&precio' = precio \\
&msg! = ok
\end{aligned}
$$

es equivalente a:

$$
\begin{aligned}
&p? \in \operatorname{dom} precio \land p? \notin \operatorname{dom} contacto \land \\
&contacto' = contacto \cup \{p? \mapsto c?\} \land precio' = precio \land msg! = ok
\end{aligned}
$$

Este predicado especifica, primero, que el producto $p?$ se encuentra en el dominio de la función *precio*, lo cual es un indicador de que $p?$ es un producto válido. Además, éste no pertenece al dominio de *contacto*, lo que determina que el producto no fue previamente reservado por otro cliente. Estos predicados indican que la operación *ReservarOk* no es una operación total, sino parcial, ya que no se especifica el comportamiento de la operación cuando no se verifican las condiciones establecidas. Luego se define cómo debe reaccionar el sistema si las condiciones anteriores se cumplen. De esta manera, se define que se agrega a la función *contacto* el par ordenado que comienza con el producto $p?$ y tiene como segundo elemento el cliente $c?$, es

decir, se agrega una reserva. El valor de *precio* no se modifica, ya que no es el objetivo de la operación. Por último, se emite el mensaje *ok* indicando que se realizó una reserva.

Como en *ReservarOk* no se especifica el comportamiento de la operación cuando $p? \notin \operatorname{dom} precio$ o cuando $p? \in \operatorname{dom} contacto$, definimos dos operaciones que tengan en cuenta estos casos, y luego definimos la operación total *Reservar*, como la disyunción de las demás:

$$ReservarE1 ==$$
$$[\Xi Reservas;\ p? : PRODUCTO;\ msg! : MENSAJE \mid$$
$$p? \notin \operatorname{dom} precio \land msg! = productoNoExistente]$$

$$ReservarE2 ==$$
$$[\Xi Reservas;\ p? : PRODUCTO;\ msg! : MENSAJE \mid$$
$$p? \in \operatorname{dom} contacto \land msg! = productoNoDisponible]$$

$$Reservar == ReservarOk \lor ReservarE1 \lor ReservarE2$$

En este caso utilizamos la notación comprimida de Z, que nos permite definir pequeños esquemas de forma más simple. El nombre se indica primero, y luego, entre corchetes, se define el esquema de la misma forma que antes, con la única excepción que se utiliza el símbolo | para separar la declaración del predicado. Estas nuevas operaciones hacen uso del operador Ξ, el cual define las mismas variables que Δ, pero indica que éstas no modifican su valor al aplicar la operación. Entonces, estas operaciones emiten como salida el mensaje indicado, pero *contacto* y *precio* no son modificadas.

Por último, definimos *Reservar* mediante una expresión de esquemas. Este tipo de expresiones permiten definir esquemas mediante el uso de otros esquemas y algunos conectores lógicos. Si bien pueden ser complejas, para este ejemplo nos alcanza con explicar que la operación comparte la sección de declaración de las tres operaciones ya definidas y tiene como predicado la disyunción de los predicados de ellas. Queda así entonces definida la opera-

ción total *Reservar*, la cual utilizaremos a continuación como ejemplo para introducir el TTF y Fastest.

2.2. El Test Template Framework

En esta sección veremos brevemente el alcance del TTF, para más detalles se puede consultar [8] [9] [10]. Hemos mencionado anteriormente que el TTF es un método que permite generar casos de prueba como parte de un proceso de MBT, el cual ha sido mayormente utilizado con especificaciones Z. Cada operación dentro de la especificación es analizada para derivar casos de prueba abstractos acordes a ella siguiendo los pasos a continuación:

- Obtener el Espacio de Entrada Válido (VIS) de cada operación Z
- Aplicar tácticas de testing en pos de particionar el VIS de cada operación
- Construir un árbol de clases de prueba
- Eliminar las clases de prueba insatisfacibles del árbol
- Obtener un caso de prueba para cada clase de prueba en el árbol

Antes de realizar el primer paso, hay que elegir los esquemas de operación sobre los cuales queremos realizar el proceso. En nuestro ejemplo, tanto *Reservar* como *ReservarOk* y las operaciones de error son esquemas de operación, pero como las últimas se utilizaron para definir la primera, solo elegiremos *Reservar*, ya que así generaremos casos también para las demás.

2.2.1. Generando el Espacio de Entrada Válido de una operación Z

El VIS de una operación se deriva de su espacio de entrada (IS). El IS de una operación es el esquema que declara todas sus variables de estado y de entrada. Por ejemplo, el IS de la operación *Reservar* es:

$$Reservar^{IS} ==$$
$$[precio : PRODUCTO \pfun VALOR;$$
$$contacto : PRODUCTO \pfun CLIENTE;$$
$$p? : PRODUCTO;\ c? : CLIENTE]$$

El VIS de una operación es el esquema que restringe el IS a verificar su precondición:

$$Op^{VIS} == [Op^{IS} \mid \text{pre } Op]$$

La precondición de una operación es la parte de su predicado que no contiene variables primadas ni de salida. En Z, esto se indica como: pre *Op*. Como *Reservar* es una operación total, su precondición es *true*, por lo tanto, su VIS es igual a su IS:

$$Reservar^{VIS} == Reservar^{IS}$$

2.2.2. Aplicando tácticas de testing

En el TTF se particiona el VIS de cada operación en clases de equivalencia, mediante la aplicación de distintas tácticas de testing. De estas clases de equivalencia, llamadas *clases de prueba*, más adelante serán derivados los casos de prueba. Formalmente, una clase de prueba S de una operación Op, es un conjunto tal que $S \subseteq Op^{VIS}$. El aspecto clave es que la sucesiva aplicación de tácticas sobre las clases de prueba genera clases cada vez más restrictivas y detalladas. Este proceso continúa hasta que la persona a cargo de la tarea decida que ha obtenido ya el nivel de precisión deseado para revelar errores en la implementación. Esta etapa es crucial dentro del proceso de generación de casos de prueba ya que determina tanto el tipo de situaciones en las que se ejercitará el código como el nivel de rigurosidad con el que se realizará el testing.

Una táctica determina cómo una clase de prueba (o el VIS) de una operación debe ser particionada, mediante la definición de una serie de predicados, los cuales son conjugados al predicado de la clase cuando la táctica es aplica-

da. Dado que cada uno de estos predicados caracteriza una clase de prueba, se los denomina *predicados característicos*. Cada táctica de testing definirá los predicados característicos de forma que las clases de prueba generadas por ellos orientan el testing a distintos aspectos del programa.

Cuando se propuso el TTF, los autores definieron algunas tácticas y algunas más fueron definidas más tarde por Cristiá [19]. A continuación aplicaremos en nuestro ejemplo algunas de las originales, y veremos más en detalle cada una de ellas en el Capítulo 3. Comenzaremos aplicando primero Forma Normal Disyuntiva (DNF), la cual fue presentada incluso antes que el TTF. El primer paso para su aplicación es escribir el predicado de la operación en DNF. Esto es, enunciar el predicado como una disyunción de conjunciones de predicados atómicos o negaciones de predicados atómicos. Luego, hay que seleccionar la precondición de cada una de las disyunciones y utilizar estos predicados como los predicados característicos de la partición.

El primer paso en la aplicación de esta táctica en nuestro ejemplo, sobre la operación *Reservar*, es sencilla debido a que ésta ya se encuentra en DNF. Se obtienen las siguientes clases de prueba:

$$Reservar_1^{DNF} == [Reservar_{VIS} \mid p? \in \mathrm{dom}\, precio \land p? \notin \mathrm{dom}\, contacto]$$
$$Reservar_2^{DNF} == [Reservar_{VIS} \mid p? \notin \mathrm{dom}\, precio]$$
$$Reservar_3^{DNF} == [Reservar_{VIS} \mid p? \in \mathrm{dom}\, contacto]$$

Esta táctica es realmente útil ya que genera casos de prueba que ejecutarán el programa en sus principales alternativas funcionales. Sin embargo, no se testeará, por ejemplo, la implementación de los operadores matemáticos utilizados. Algunos operadores matemáticos complejos que ofrece Z, como ser $\cup$, pueden no tener una implementación simple y directa, por lo cual es importante ponerlos a prueba. El TTF ofrece para estos casos, la táctica llamada Particiones Estándar (SP). Esta táctica propone distintas particiones para cada uno de los operadores matemáticos utilizados en Z. Por ejemplo, podemos ver en la Figura 2.1, la partición para los operadores $\cup$, $\cap$ y $\setminus$ [8]. De esta manera, la táctica, parametrizada por un operador matemático, determina los predicados característicos para realizar la partición de las clases

- $A = \{\}, B = \{\}$
- $A = \{\}, B \neq \{\}$
- $A \neq \{\}, B = \{\}$
- $A \neq \{\}, B \neq \{\}, A \cap B = \{\}$
- $A \neq \{\}, B \neq \{\}, A \subset B$
- $A \neq \{\}, B \neq \{\}, B \subset A$
- $A \neq \{\}, B \neq \{\}, B = A$
- $A \neq \{\}, B \neq \{\}, A \cap B \neq \{\}, \neg\,(A \subseteq B), \neg\,(B \subseteq A), B \neq A$

Figura 2.1: Partición estándar para $A \cup B$, $A \cap B$ y $A \setminus B$

de prueba.

Entonces, la persona encargada de aplicar las tácticas de testing deberá primero seleccionar un operador matemático que considere que puede tener una implementación dificultosa. Y en caso que este operador se utilice en distintas ocasiones dentro de la misma clase de prueba, deberá elegir una de las apariciones. Con esa información se pueden generar, a partir de la partición propuesta para el operador, las nuevas clases de prueba. Luego de eso, el usuario, puede volver a aplicar la táctica, por ejemplo, sobre las otras apariciones del mismo operador o sobre otros operadores si así lo desea.

Vamos a aplicar SP en nuestro ejemplo sobre el operador $\cup$ presente en $contacto \cup \{p? \mapsto c?\}$. Sólo aplicaremos la táctica sobre $Reservar_1^{DNF}$ ya que, muy posiblemente, se utilizará la implementación del operador en ese caso y no cuando no se pueda realizar la reserva, como en $Reservar_2^{DNF}$ y $Reservar_3^{DNF}$. La partición para $contacto \cup \{p? \mapsto c?\}$ propuesta genera ocho

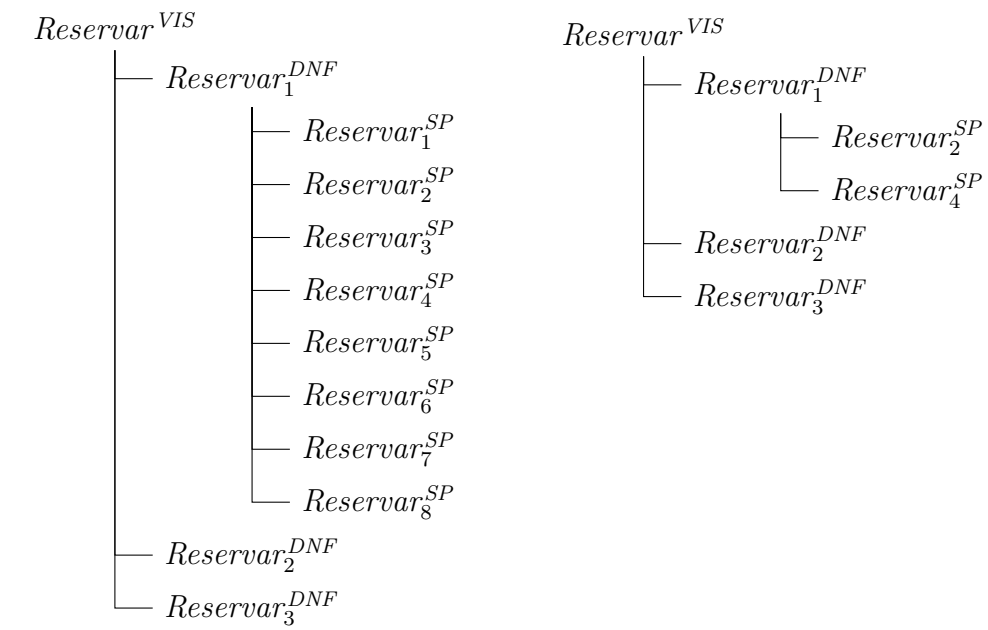
nuevas clases:

$$
\begin{aligned}
&Reservar_1^{SP} == [Reservar_1^{DNF} \mid contacto = \{\} \land \{p? \mapsto c?\} = \{\}] \\
&Reservar_2^{SP} == [Reservar_1^{DNF} \mid contacto = \{\} \land \{p? \mapsto c?\} \neq \{\}] \\
&Reservar_3^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} = \{\}] \\
&Reservar_4^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} \neq \{\} \land \\
&\quad contacto \cap \{p? \mapsto c?\} = \{\}] \\
&Reservar_5^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} \neq \{\} \land \\
&\quad contacto \subset \{p? \mapsto c?\}] \\
&Reservar_6^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} \neq \{\} \land \\
&\quad \{p? \mapsto c?\} \subset contacto] \\
&Reservar_7^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} \neq \{\} \land \\
&\quad \{p? \mapsto c?\} = contacto] \\
&Reservar_8^{SP} == [Reservar_1^{DNF} \mid contacto \neq \{\} \land \{p? \mapsto c?\} \neq \{\} \land \\
&\quad contacto \cap \{p? \mapsto c?\} \neq \{\} \land \neg\, contacto \subseteq \{p? \mapsto c?\} \land \\
&\quad \neg\, \{p? \mapsto c?\} \subseteq contacto \land \{p? \mapsto c?\} \neq contacto]
\end{aligned}
$$

Como se puede notar, la aplicación de la táctica se reduce a reemplazar los parámetros formales en la partición por los reales encontrados en el esquema seleccionado. En este ejemplo, también puede aplicarse SP en otras expresiones, pero no lo haremos para mantener el ejemplo sencillo, aunque en la práctica real debería hacerse. Veremos la aplicación de otras tácticas en detalle en el Capítulo 3.

2.2.3. Construcción del árbol de clases

El TTF ordena las clases de prueba de una determinada operación, en lo que se denomina un árbol de clases de prueba. La construcción de este árbol comienza por el VIS de la operación como su raíz. La aplicación de una táctica sobre un nodo, genera nuevas clases que serán nodos hijos del mismo. Entonces, la sucesiva aplicación de tácticas aumenta nivel por nivel el tamaño del árbol. El predicado de cada nuevo nodo es construido como la conjunción del predicado de su padre y el predicado característico de la táctica aplicada. En la Figura 2.2a podemos ver el árbol para la operación



(a) Original (b) Podado

Figura 2.2: Árboles de clases original y podado de *Reservar*

Reservar construido anteriormente como ejemplo.

Esta forma de ordenar las clases de prueba permite limitar la búsqueda de casos de prueba a las hojas del árbol, dado que si un caso determinado satisface un nodo, también lo hará para su padre.

2.2.4. Podando el árbol

Algunas de las clases de prueba del árbol pueden resultar vacías porque sus predicados no son satisfacibles. En estos casos, hay que podar estos nodos, ya que no es posible encontrar casos de prueba para ellos. En la Figura 2.2b, la mayoría de las hojas fueron podadas del árbol, como por ejemplo, $Reservar_1^{SP}$, ya que el conjunto con el elemento $p? \mapsto c?$ no es vacío, o en $Reservar_7^{SP}$, $\{p? \mapsto c?\} = contacto$ no puede verificarse si $p? \notin \text{dom}\, contacto$.

2.2.5. Derivando casos de prueba

El último paso propuesto por el TTF es encontrar valores para las variables que satisfagan las clases de prueba. Estos valores toman el nombre de casos de prueba abstractos, y son los que se utilizan para generar los casos con los que testear la implementación. En nuestro ejemplo podemos encontrar los siguientes:

$$
\begin{aligned}
&Reservar_1^{TC} == [Reservar_2^{SP} \mid p? = producto1 \land \\
&\quad precio = \{(producto1 \mapsto 2)\} \land c? = cliente3 \land contacto = \emptyset] \\
&Reservar_2^{TC} == [Reservar_4^{SP} \mid p? = producto1 \land \\
&\quad precio = \{(producto1 \mapsto 2)\} \land c? = cliente5 \land \\
&\quad contacto = \{(producto3 \mapsto cliente4)\}] \\
&Reservar_3^{TC} == [Reservar_2^{DNF} \mid p? = producto2 \land \\
&\quad precio = \emptyset \land c? = cliente1 \land contacto = \emptyset] \\
&Reservar_4^{TC} == [Reservar_3^{DNF} \mid p? = producto1 \land \\
&\quad precio = \emptyset \land c? = cliente3 \land contacto = \{(producto1 \mapsto cliente2)\}]
\end{aligned}
$$

Notar cómo los casos de prueba también se relacionan con las clases de prueba mediante la inclusión de esquemas.

2.3. Sobre Fastest

Fastest es una herramienta que implementa gran parte del proceso de generación de casos de prueba propuesto por el TTF. Fue concebida por Cristiá y desarrollada en Java, en su mayoría por Monetti y Albertengo [14] [15] [16]. Actualmente implementa las etapas del TTF enunciadas en las secciones anteriores, pero el proyecto también incluye el refinamiento y abstracción de los casos de prueba, para poder luego ejecutar el programa en esas situaciones y verificar los resultados de forma automática. La herramienta se ha logrado usar exitosamente en algunos casos de estudio de nivel industrial [17].

Previamente vimos cómo generar los casos de prueba siguiendo los procedimientos descriptos por el TTF. A continuación, tomaremos el mismo ejemplo en Z, y generaremos los casos de prueba utilizando la herramien-

ta. El primer paso a realizar, luego de ejecutar la herramienta, es cargar la especificación Z:

```
loadspec sistemaDeReservas.tex
```

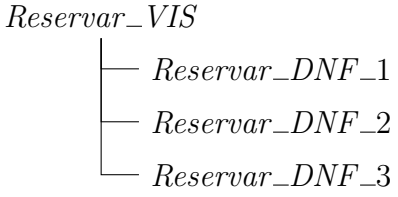
El comando `loadspec` se utiliza para indicar a Fastest el nombre del archivo que contiene la especificación a utilizar, en este caso, `sistemaDeReservas.tex`. Esta especificación debe estar escrita en LaTeX utilizando el paquete *z-eves* y siguiendo el estándar ISO. Fastest realizará un chequeo sintáctico y de tipos una vez que se cargue la especificación, indicando si hay errores. Si la especificación es correcta, debemos entonces seleccionar los esquemas de operación para los cuales buscaremos los casos de prueba:

```
selop Reservar
```

En este ejemplo, buscamos casos únicamente para la operación *Reservar*. Esto lo indicamos mediante el comando `selop`. En caso de querer buscar casos de prueba para varias operaciones, se pueden invocar varios comandos `selop`. Una vez indicada la operación, podemos comenzar a aplicar las tácticas de testing. Fastest aplica, siempre, DNF como primera táctica. Por lo tanto, si no indicamos ninguna táctica a aplicar, se utilizará únicamente DNF:

```
genalltt
```

La aplicación de tácticas en Fastest consta de dos pasos básicos, el primero es la elección de las tácticas y sus argumentos, y el segundo la generación del árbol de clases usando esas tácticas. `genalltt` es el comando que construye el árbol. Como en nuestro caso no seleccionamos ninguna táctica, obtenemos sólo la aplicación de DNF sobre *Reservar*. El resultado puede ser observado mediante el comando `showtt`:

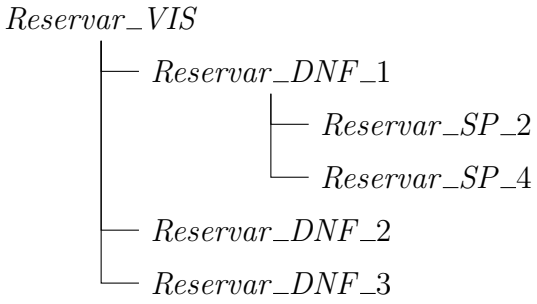


En nuestro ejemplo, queremos aplicar la táctica SP únicamente sobre *Reservar_DNF_1*. Para ello, utilizamos el comando `addtactic`:

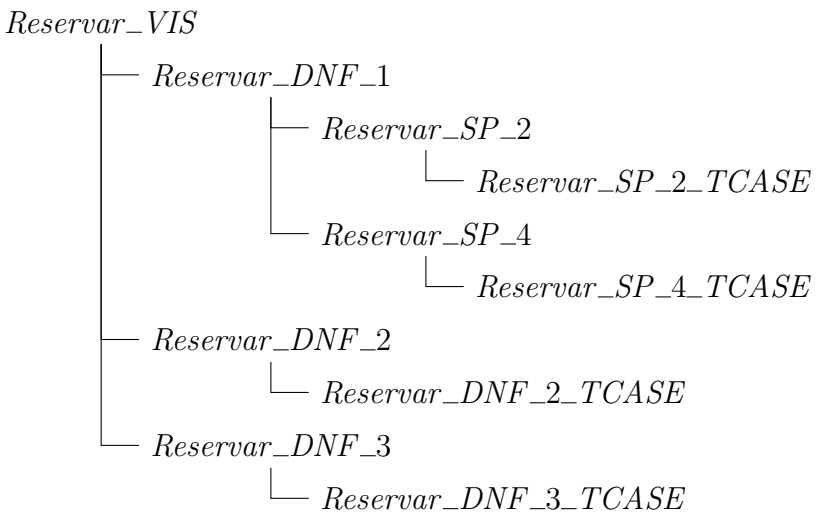
```
addtactic Reservar_DNF_1 SP \cup contacto \cup \{p? \mapsto c?\}
```

Este comando toma como entrada cuatro parámetros. El primero, en este caso `Reservar_DNF_1`, es el nombre de la clase sobre la cual queremos aplicar la táctica. El segundo parámetro es el nombre de la táctica, `SP`. El siguiente argumento indica el operador Z sobre el que se quiere trabajar. De esta manera Fastest puede determinar la partición a utilizar. Por último, se indica la expresión Z sobre la que se aplica la táctica, de manera que se pueden determinar los argumentos de la partición. Notar que los últimos argumentos deben ser indicados utilizando la notación Z en LaTeX. Una vez indicada la táctica, podemos generar el resto del árbol usando nuevamente `genalltt`.

Como vimos anteriormente, algunas de las clases generadas no serán satisfacibles. Fastest permite podar los nodos del árbol de forma manual, o se puede usar el comando `prunett` para que intente hacerlo de forma automática. Utilizamos entonces este comando, lo que deja como resultado el siguiente árbol del cual algunas hojas fueron podadas:



Como último paso, debemos derivar los casos de prueba. Fastest intenta encontrar un caso para cada clase utilizando el comando `genalltca`. Este, buscará un caso para cada una de las clases que sean hojas en el árbol. Si no logra encontrar un caso, el usuario podrá especificarlo manualmente si así lo desea. No explicaremos cómo la herramienta busca casos de prueba ya que no está relacionado con el aporte de este trabajo, además ya ha sido explicado [13] [14] [18]. Para nuestro ejemplo, Fastest encuentra los siguientes:



No los mostraremos en detalle ya que son los que vimos anteriormente al describir el TTF.

Capítulo 3

Tácticas de Testing

En este capítulo veremos en detalle las tácticas disponibles en Fastest, con el fin de analizar los casos en los que se debe aplicar cada una y la forma en que hay que hacerlo. Si bien en el TTF se definen más tácticas, nos centraremos únicamente en algunas de las que fueron implementadas por la herramienta, ya que son las que utilizaremos luego para definir las estrategias de testing. Para más información pueden verse [13] [19].

3.1. Forma Normal Disyuntiva

Esta táctica fue definida originalmente por Stocks y Carrington, aunque proviene en realidad del concepto de la lógica que le aporta su nombre. En Fastest es la primer táctica que se utiliza para construir el árbol. Para eso, primero debe definir la forma normal disyuntiva del predicado de la operación a testear, esto es, expresar el predicado como una disyunción de conjunciones de literales. Se entiende por literal a un predicado atómico o la negación de un predicado atómico. El primer paso para llevar cualquier predicado a DNF es transformar todas las equivalencias de la forma $a \Leftrightarrow b$ en el predicado $(a \Rightarrow b) \wedge (b \Rightarrow a)$. Modificadas todas las equivalencias, debemos transformar las implicaciones, expresando los predicados $a \Rightarrow b$ de la forma $\neg\ a \vee b$. Como tercer paso, se deben distribuir las negaciones ($\neg$) de forma de generar únicamente negaciones de predicados atómicos. Para ello se utilizan las

siguientes reglas:

- $\neg \neg a \Leftrightarrow a$
- $\neg (a \vee b) \Leftrightarrow \neg a \wedge \neg b$
- $\neg (a \wedge b) \Leftrightarrow \neg a \vee \neg b$

Por último, debemos distribuir las conjunciones sobre las disyunciones. Como resultado obtenemos el predicado original expresado en DNF. Entonces, podemos tomar cada una de las disyunciones que formen parte de la precondición de la operación y generar una nueva clase de prueba con cada una de ellas. En el Capítulo 2, aplicamos esta táctica sobre un ejemplo sencillo, sobre la operación *Reservar*. En ese caso, como la operación se define a partir de la disyunción de los esquemas *ReservarOk*, *ReservarE*1 y *ReservarE*2, y como estos esquemas ya están expresados en DNF, la operación total ya se encuentra expresada en DNF. Por lo tanto, la aplicación de la táctica se reduce a generar una clase por cada una de estas operaciones.

El objetivo de esta táctica es generar casos de test para cada una de las alternativas lógicas especificadas para las operaciones. Por ejemplo, para las operación *Reservar*, se genera un caso para cuando el producto que se desea reservar no es un producto del negocio; otro caso cuando el producto ya se encuentra reservado y un tercer caso para cuando se puede realizar la reserva. Estos casos de prueba son propensos a encontrar errores en la implementación bajo la suposición que en el código se tratan de manera distinta estos casos, así como se lo hizo en la especificación. De esta manera, se ejercitan ambas secciones del código, tanto la que maneja los casos exitosos como la que lo hace para los erróneos. Una posible implementación, podría ser por ejemplo:

```
if (p? ∈ dom precio ∧ p? ∉ dom contacto) then
    contacto' = contacto ∪ {p? ↦ c?}
    msg! = ok
else if (p? ∉ dom precio)
    msg! = productoNoExistente
else
    msg! = productoNoDisponible
```

de manera que se testearía el código en cada una de las tres situaciones posibles.

Un punto a notar es que esta táctica puede no generar una partición. Eso dependerá de cómo se haya definido la operación en cuestión. Si bien este no es un comportamiento deseable, en el sentido que se pueden generar casos de prueba repetidos u orientados al testing de la misma situación, podemos asegurarnos que habrá al menos un caso por cada una de ellas.

En Fastest la aplicación de esta táctica se consigue utilizando el comando `genalltt`. Como se ha indicado antes, este comando además realiza la aplicación de otras tácticas si es que hay alguna seleccionada. Entonces, DNF es aplicada de forma distinta al resto de las tácticas en Fastest. Mientras todas las tácticas deben ser seleccionadas previamente para ser aplicadas luego con `genalltt`, DNF es aplicada automáticamente la primera vez que se llama al comando.

3.2. Particiones Estándar

Debido a que las especificaciones Z no son únicamente una fórmula lógica, sino que hacen fuerte uso de operadores matemáticos (especialmente aquellos de la teoría de conjuntos), la aplicación de DNF no es suficiente para crear casos de prueba complejos. El TTF introduce entonces la táctica *Particiones Estándar* (SP, de *Standard Partitions* en inglés).

Muchas veces los desarrolladores prefieren utilizar, por ejemplo, listas como representación de conjuntos finitos, en vez de utilizar estructuras de datos ya provistas por lenguajes como Java (como HashSet). Debido a esto, la implementación de algunos operadores matemáticos puede no ser trivial. Para estos casos, es recomendable utilizar la táctica SP, la cual, como vimos anteriormente, fue pensada para generar casos de prueba orientados a testear el código en las distintas situaciones en las que se ve involucrada la implementación de este tipo de operadores.

Para esto, el TTF define distintas particiones del dominio de una operación, en función del operador que se quiera analizar. Fastest, a su vez, define estas particiones y agrega algunas nuevas. Estas particiones fueron

pensadas para ser modificadas por el usuario según sus necesidades, dando la posibilidad de generar nuevas particiones para otros operadores también. Actualmente se proveen particiones para operadores como $\notin$, $\lhd$, $\oplus$, $\setminus$, $\cup$ y $\#$, entre otros tantos.

Si tenemos por ejemplo, el predicado $A' = A \cup B$ como parte del predicado de nuestra operación, la partición definida para el operador $\cup$ determina que se generarán clases para los siguientes casos:

- Tanto A como B son conjuntos vacíos
- A es vacío, pero B no lo es
- A no es vacío, pero lo es B
- Ninguno es vacío, pero su intersección lo es
- Ninguno es vacío y A es un subconjunto de B
- Ninguno es vacío y B es un subconjunto de A
- Ninguno es vacío y A es igual a B
- A y B tienen elementos en común, pero ninguno es un subconjunto del otro

Para aplicar esta táctica en Fastest, el usuario deberá utilizar el comando `addtactic` indicando algunos argumentos:

- Nombre del esquema o nodo, por ejemplo *Reservar*
- Nombre de la táctica (en este caso SP)
- Nombre del operador en LaTeX, por ejemplo `\cup` ($\cup$)
- Expresión objetivo en LaTeX, por ejemplo `A \cup B` ($A \cup B$)

Los primeros dos argumentos son requeridos en la aplicación de cualquiera de las tácticas (excepto DNF que se aplica automáticamente). El nombre del esquema o nodo, indica sobre qué clase de prueba se aplica la táctica.

Si se utiliza el nombre de una operación, como ser *Reservar*, la táctica se aplicará sobre todas las hojas del árbol; en cambio, si se selecciona un nodo del árbol, la táctica será aplicada sobre las hojas del subárbol que tiene por raíz el nodo en cuestión. El otro argumento en común es el nombre de la táctica a utilizar, en este caso se indica con *SP*.

El resto de los parámetros son particulares de la táctica seleccionada. Al aplicar SP debemos indicar, tanto el operador matemático a utilizar, como la expresión donde aparece. Ambos argumentos deberán expresarse en LaTeX, tal como aparecen en la especificación Z. Fastest buscará entonces la partición correspondiente para el operador, y reemplazará en ésta los argumentos utilizados por las expresiones reales. De esta manera obtiene los predicados característicos con los que genera las nuevas clases de prueba. Estas serán agregadas al árbol de clases como hijos de cada uno de los nodos hoja sobre los que se aplicó la táctica. De esta manera, se crean nuevos nodos hoja, aumentando la profundidad del árbol en el proceso.

3.3. Rangos Numéricos

Si bien DNF y SP proporcionan una buena variedad de casos de prueba, algunos casos interesantes aún quedan por fuera. Por este motivo, en Fastest, se definió la táctica *Rangos Numéricos* (NR, del inglés *Numeric Ranges*). Esta táctica tiene como parámetro una expresión numérica *expr* y un rango de valores numéricos de la forma $\langle v_1, ..., v_n \rangle$, donde se verifica $v_1 < ... < v_i < ... < v_n$. La táctica toma estos valores y genera las siguientes particiones:

- $expr < v_1$
- $v_i < expr < v_{i+1}$, para cada i entre 1 y $n - 1$
- $v_n < expr$
- $expr = v_i$, para cada i entre 1 y n

Esta táctica es muy útil, por ejemplo, para testear cómo se comportan los programas cuando sus variables numéricas se encuentran dentro o fuera de los límites creados por su implementación. Por ejemplo, una variable del tipo `short` en *C*, puede adoptar valores en el rango $[SHRT\_MIN, SHRT\_MAX]$. Entonces, es razonable querer testear el programa con valores mayores, menores e iguales a $SHRT\_MIN$ y $SHRT\_MAX$, para aquellas variables de estado o de entrada que han sido implementadas bajo esas limitaciones.

Aquellas variables definidas en Z utilizando el tipo $\mathbb{N}$, muy probablemente sean luego representadas en la implementación utilizando un tipo numérico como `short` o `int`, entre otros, los cuales presentan límites tanto superiores como inferiores en los valores que pueden adoptar las variables.

Si observamos el ejemplo definido en el capítulo anterior, una posible implementación en C para nuestra función de precios es utilizar una lista simplemente enlazada como:

```
struct precio_nodo
   {char* producto;
    short valor;
    struct precio_nodo* proximo}
```

Por otro lado, podemos definir una operación simple para modificar el precio de un producto:

ModificarValorOk

$\Delta Reservas$
$p? : PRODUCTO$
$v? : VALOR$

$p? \in \operatorname{dom} precio$
$precio' = precio \oplus \{p? \mapsto v?\}$

Entonces, si se desea observar el comportamiento de la operación cuando un `producto` tiene un `valor` en los límites del rango de valores de `short`, se puede aplicar la táctica NR, por ejemplo sobre *ModificarValorOk*:

```
addtactic ModificarValorOk NR v? [-32768, 32767]
```

la cual generará las siguientes clases de prueba:

$$ModificarValorOk_1^{NR} == [ModificarValorOk_1^{DNF} \mid v? < -32768]$$
$$ModificarValorOk_2^{NR} == [ModificarValorOk_1^{DNF} \mid v? = -32768]$$
$$ModificarValorOk_3^{NR} == [ModificarValorOk_1^{DNF} \mid v? > -32768 \land v? < 32767]$$
$$ModificarValorOk_4^{NR} == [ModificarValorOk_1^{DNF} \mid v? = 32767]$$
$$ModificarValorOk_5^{NR} == [ModificarValorOk_1^{DNF} \mid v? > 32767]$$

Notar que para lograr una óptima aplicación de la táctica, algunos detalles de la implementación son requeridos. Este tema es abordado con más detalle en el Capítulo 4.

3.4. Tipos Libres

Tipos Libres (FT, del inglés *Free Types*) es una táctica de testing pensada para las situaciones donde se utilizan variables de tipo libre. Los tipos libres en Z son aquellos tipos donde se especifican exactamente los valores que puede tomar una variable del tipo. Si se selecciona una de estas variables para utilizar la táctica, ésta generará una clase por cada uno de los posibles valores en el tipo.

Por ejemplo, podemos definir el tipo *ESTADO* que utilizaremos para determinar si el producto debe ser enviado o si el cliente lo retirará en el comercio:

$$ESTADO ::= enviar \mid retirar$$

Para esto, podemos agregar la función *envio* en el esquema *Reservas*:

Reservas

$$precio : PRODUCTO \pfun VALOR$$
$$contacto : PRODUCTO \pfun CLIENTE$$
$$envio : PRODUCTO \pfun ESTADO$$

y debemos agregar también una variable $e?$, como variable de entrada en el esquema *ReservarOk*, donde el cliente indicará cómo desea coordinar la entrega:

$$
\begin{array}{|l}
\hline
\textit{ReservarOk} \\
\Delta \textit{Reservas} \\
e? : \textit{ESTADO} \\
... \\
\hline
\textit{contacto}' = \textit{contacto} \cup \{p? \mapsto c?\} \\
\textit{envio}' = \textit{envio} \cup \{p? \mapsto e?\} \\
... \\
\hline
\end{array}
$$

De esta manera, podríamos querer ver el comportamiento del sistema cuando se realiza la reserva de un producto que se debe enviar, o cuando no. Entonces la táctica generará una clase de prueba para cada uno de los posibles valores de *ESTADO*:

$$\textit{ReservarOk}_1^{FT} == [\textit{ReservarOk}_1^{DNF} \mid e? = \textit{enviar}]$$
$$\textit{ReservarOk}_2^{FT} == [\textit{ReservarOk}_1^{DNF} \mid e? = \textit{retirar}]$$

Quedan así generados casos que cubrirán el testing tanto en las situaciones donde el producto se retirará por el negocio, como cuando debe ser enviado a la dirección del cliente. En Fastest, para aplicar la táctica se debe especificar únicamente el nombre de la variable, además de los parámetros necesarios del comando:

```
addtactic ReservarOk_DNF_1 FT e?
```

3.5. Conjuntos por Extensión

Esta táctica fue definida en Fastest, bajo el nombre en inglés de *In Set Extension* (ISE), y puede aplicarse sobre los esquemas de operación que incluyen, en su precondición, predicados de la forma:

$expr \in \{expr_1, ..., expr_n\}$

En este caso genera $n+1$ clases de prueba con las condiciones siguientes:

- $expr \notin \{expr_1, ..., expr_n\}$
- $expr = expr_i$, para cada i entre 1 y n

Para la utilización en Fastest de la táctica, el único parámetro que se debe proporcionar es la expresión con el predicado de esa forma:

```
addtactic Op ISE expr \in \{expr_1, ..., expr_n\}
```

Capítulo 4

Estrategias de Testing

La aplicación de tácticas de testing no es una actividad dificultosa, pero requiere que el usuario tenga conocimientos del TTF para determinar las tácticas que desea aplicar y conocimientos de Z para poder analizar la especificación y buscar los parámetros que se utilizarán en ellas. A su vez, aún cuando se tengan estos conocimientos, esta actividad puede demandar mucho tiempo cuando se trabaja con especificaciones de tamaño considerable.

En el Capítulo 2 vimos un pequeño ejemplo, donde la construcción del árbol de clases de prueba se limitaba a la aplicación de dos tácticas de testing. En el mismo escenario, si el usuario quisiera realizar un testing más exhaustivo de la operación, se vería obligado a aplicar SP en más situaciones, por ejemplo, podría haber utilizado los siguientes comandos en Fastest:

```
genalltt
addtactic Reservar_DNF_1 SP \cup contacto \cup \{p?\mapstoc?\}
addtactic Reservar_DNF_1 SP \in p? \in \dom precio
addtactic Reservar_DNF_1 SP \notin p? \notin \dom contacto
addtactic Reservar_DNF_2 SP \notin p? \notin \dom precio
addtactic Reservar_DNF_3 SP \in p? \in \dom contacto
genalltt
```

donde el primero construye el árbol inicial aplicando DNF y el resto aplica SP sobre las distintas expresiones encontradas en cada operación.

Estrategias	Tácticas
FUNCIONALIDAD BASICA	DNF
ENUMERACIONES	DNF FT
LIMITES DE IMPLEMENTACION	DNF NR
VALORES IMPORTANTES	DNF ISE
MATEMATICA	DNF SP
TODA FUNCIONALIDAD	DNF SP FT
MATEMATICA Y LIMITES	DNF SP NR
BUEN TESTING	DNF SP NR ISE
TESTING FUERTE	DNF SP NR FT
TESTING COMPLETO	DNF SP NR FT ISE

Figura 4.1: Tácticas de testing usadas en las estrategias de testing

Queremos que los usuarios tengan la posibilidad de omitir el análisis requerido para utilizar esos comandos. Entonces, proponemos la definición e implementación en Fastest de lo que llamamos *estrategias de testing*, las cuales permiten, por ejemplo, reemplazar la lista de comandos anterior por la utilización de una sola directiva notablemente más simple:

```
applystrategy Enumeraciones Reservar [intensidad]
```

Una estrategia de testing combina diferentes tácticas de testing para ser aplicadas de forma automática, con la menor interacción posible por parte del usuario. Lo que permite que el trabajo de éste se limite a la elección de la estrategia que desea aplicar. Siguiendo el ejemplo, el usuario indica únicamente que desea aplicar la estrategia ENUMERACIONES sobre la operación *Reservar* y Fastest aplica las tácticas DNF y SP toda las veces que puede[1].

A continuación enumeramos las estrategias que hemos definido en Fastest, con una pequeña descripción de cada una de ellas en base al tipo de testing y casos de prueba que generan. En la tabla 4.1 se puede ver el conjunto de tácticas de testing que aplica cada una.

[1]La cantidad de veces que se aplica cada táctica depende de la intensidad de testing elegida. Este parámetro es opcional y será explicado más adelante.

- FUNCIONALIDAD BASICA

  Esta es la más simple de las estrategias y la que genera la menor cantidad de casos de prueba. Únicamente genera casos en base a las alternativas lógicas presentes en la operación.

- ENUMERACIONES

  Como lo indica su nombre, esta estrategia se centra en la generación de casos de prueba en base a los posibles valores de las variables de tipo enumerado encontradas en la operación.

- LIMITES DE IMPLEMENTACION

  Cuando los aspectos del código que se desean testear son las limitaciones inducidas por el uso de variables de tipo numérico, esta es la estrategia recomendada.

- VALORES IMPORTANTES

  Algunas operaciones indican explícitamente los valores que puede tener una determinada variable. Cuando se desea testear la operación en cuestión con todos estos valores se debe usar esta estrategia.

- MATEMATICA

  Esta es una de las estrategias más utilizadas, ya que genera casos de prueba orientados al testing de la implementación de ciertos operadores matemáticos.

- TODA FUNCIONALIDAD

  Esta estrategia combina las estrategias MATEMATICA y ENUMERACIONES. De esta manera, genera clases de prueba tanto para el testing de la implementación de operadores matemáticos, como para verificar el comportamiento de la operación cuando las variables de tipo enumerado toman distintos valores.

- MATEMATICA Y LIMITES

  En este caso se combinan MATEMATICA y LIMITES DE IMPLEMENTACION, generando casos tanto para el testing de la implementación de opera-

dores matemáticos, como para ejercitar la implementación cuando las variables de tipo numérico adquieran valores por fuera de sus límites.

- BUEN TESTING

  Esta estrategia agrupa las principales tácticas utilizadas, lo que genera un buen cubrimiento de testing en una gran cantidad de situaciones. Debido a que puede generar una gran cantidad de clases de prueba, se recomienda utilizarla para el testing de operaciones críticas o utilizar una intensidad de testing relativamente baja. Al igual que MATEMATICA Y LIMITES se centra en testear la operación haciendo foco en la implementación de los operadores matemáticos y los tipos numéricos utilizados. Además, busca generar casos para los principales valores que pueden tomar ciertas variables.

- TESTING FUERTE

  Al igual que BUEN TESTING, esta estrategia genera un buen cubrimiento de testing. La única diferencia es que esta estrategia utiliza los valores propuestos para las variables de tipo enumerado presentes en la operación, lo que suele generar casos más útiles a los generados por la estrategia BUEN TESTING.

- TESTING COMPLETO

  En este caso se aplican todas las tácticas, lo que resulta en el mejor cubrimiento posible para la operación si se utiliza con la mayor intensidad de testing.

Como es posible notar, intentamos nombrar las estrategias con identificadores que señalen el tipo de testing que producirán, en vez del conjunto de tácticas que utiliza cada una. Esto se debe a que nuestra intención es esconder al usuario estos detalles, pero a su vez, permitirle a los usuarios avanzados aprender la composición de cada una al explorar la documentación.

Desde la perspectiva del usuario, las estrategias de testing se diferencian unas de otras por las clases de prueba que generan. Tomando el trabajo de Rapps y Weyuker [20] como inspiración, organizamos las estrategias de testing de acuerdo a un orden parcial como se muestra en el diagrama de

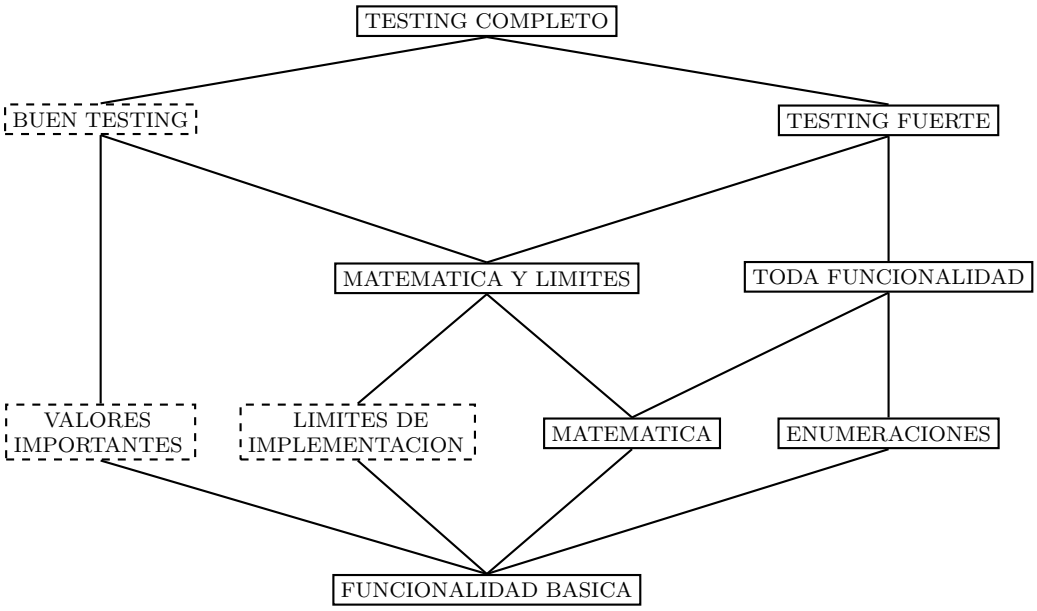


Figura 4.2: Orden parcial entre estrategias

Hasse representado en la figura 4.2. Las estrategias más cercanas a la parte superior del diagrama son las que producen una mejor cobertura y aquellas más próximas al mínimo producen una peor cobertura.

Debido a la manera en que las tácticas de testing se combinan para construir el árbol de clases de prueba en el TTF (por ejemplo mediante la inclusión de esquemas, lo cual es una forma de conjunción lógica), no es importante el orden en el cual se aplican las tácticas. Consideremos la aplicación de dos tácticas cualesquiera $T1$ y $T2$ sobre una determinada operación Op, donde la aplicación de cada una de ellas produce n y m nodos respectivamente si son aplicadas de forma individual. Entonces, si ambas tácticas se aplican sobre Op, sin importar el orden en que se realice, se generarán $n \cdot m$ nuevos nodos. Esto también explica el orden parcial utilizado para construir el diagrama de la figura 4.2. En efecto, si E_1 y E_2 son dos estrategias de testing tales que hay una arista ascendente de E_1 a E_2, entonces E_2 puede producir al menos el mismo conjunto de clases de prueba que E_1. Esto se debe a que

todas las tácticas de testing que son aplicadas en E_1 son aplicadas también en E_2, más algunas más. Estas tácticas extras no necesariamente producirán nuevas clases de prueba, aunque sí lo harán en muchos casos. Por ejemplo, TODA FUNCIONALIDAD no producirá resultados diferentes a MATEMATICA si no hay variables de tipo enumerados en la operación, pero si las hay, entonces producirá más clases de prueba. Si hay una arista que asciende de E_1 a E_2 decimos que E_2 es más fuerte que E_1 o que E_1 es más débil que E_2.

Las líneas de puntos que rodean un nodo en el diagrama, indican estrategias que pueden ser definidas pero pueden no ser muy útiles. Por ejemplo, LIMITES DE IMPLEMENTACION aplica NR que prueba características de la implementación que usualmente no son tan importantes como aquellas probadas por FT o SP. Por lo tanto, no tiene demasiado sentido utilizar LIMITES DE IMPLEMENTACION en vez de, por ejemplo, MATEMATICA Y LIMITES ó simplemente MATEMATICA.

Todas las estrategias de testing reciben un parámetro que llamamos *intensidad de testing*. La intensidad de testing es un número natural positivo, el cual tiene por defecto el valor 1. Si los usuarios, cuando llaman a una determinada estrategia, utilizan un valor más grande, entonces la estrategia puede generar más clases de prueba. La cantidad de clases de prueba depende del número de expresiones de un tipo particular que están presentes en la operación Z a particionar. En la tabla 4.3 se muestran el tipo de expresiones a considerar en cada una de las tácticas de testing. Si una estrategia es definida por más de una táctica, el producto de la cantidad de expresiones de cada una define la máxima intensidad de testing para la estrategia. Por ejemplo, si *Op* es una operación Z que tiene n expresiones con un operador matemático para el cual SP esta definida, y m variables de tipo enumerado, entonces la máxima intensidad de testing para TODA FUNCIONALIDAD cuando es aplicada a *Op* es $m + n$. Entonces, si el usuario llama a TODA FUNCIONALIDAD para ser aplicada sobre *Op* con una intensidad de testing k (donde $1 < k \leq n + m$), Fastest aplicará SP y FT, k veces cada una (en diferentes expresiones y en diferentes variables de tipo enumerado), si es posible (sino, aplicará más veces una táctica hasta sumar k aplicaciones). Por otro lado, si TODA FUNCIONALIDAD se llama con una intensidad de 1, Fastest aplicará SP

Táctica	Expresiones
FT	variables de tipo enumerado
SP	operadores matemáticos y expresiones donde aparecen
NR	expresiones aritméticas
ISE	expresiones de la forma $expr \in \{expr_1, ..., expr_n\}$

Figura 4.3: Expresiones consideradas en las tácticas de testing

y FT una vez cada una, para dos expresiones elegidas entre las disponibles. Estas expresiones son elegidas en base a nuestros conocimientos empíricos sobre la efectividad de las tácticas para encontrar errores. De esta manera, se sugiere que operadores tales como $\oplus$ y $\lhd$ deben ser favorecidos sobre otros como $\cup$ y $\cap$, que expresiones que involucran subexpresiones más largas se deben elegir antes que aquellas con subexpresiones más simples, que tipos enumerados más grandes o conjuntos por extensiones más grandes deben ganar sobre los más pequeños y así sucesivamente. Los usuarios pueden consultar la máxima intensidad con respecto a una determinada operación Z.

Como puede verse, para lograr aplicar las estrategias, Fastest realiza un análisis estático de la operación seleccionada. Luego de la descripción de las tácticas de testing vistas en el Capítulo 3 está claro que toda la información necesaria para aplicarlas puede ser encontrada en la especificación, con excepción de NR. En efecto, si la intención es utilizar NR para testear cómo se comporta la implementación cuando algunas variables alcanzan los límites de su tipo, entonces Fastest necesita un poco de información sobre la implementación, la cual no es, y no debería ser, parte de la especificación Z. Algunos de los datos que Fastest podría necesitar para aplicar NR incluyen (pero no están limitados a) el lenguaje de programación, la plataforma donde la implementación va a correr, el compilador, la cantidad de memoria física que se tendrá disponible, etc. En [21] se describe un lenguaje especial que tiene como objetivo ayudar a refinar casos de prueba generados por Fastest a los lenguajes de programación de la implementación. Usuarios de este lenguaje definen las llamadas reglas de refinamiento, donde indican cómo las variables de la especificación son implementadas. De esta manera, cuando

una estrategia de testing necesita aplicar NR, mira la regla de refinamiento correspondiente y obtiene el tipo (de implementación) de la estructura de datos utilizada en la implementación de la operación Z. Con esta información, más el código fuente y algunos archivos de configuración que brinda el usuario[2], la estrategia puede realizar una implementación inteligente de NR.

Consideremos el ejemplo visto anteriormente en el Capítulo 2 y el ejemplo elaborado cuando se introdujo NR en el Capítulo 3. La regla de refinamiento determina que cada elemento en el rango de *precio* tiene un campo llamado *valor* en un nodo de la lista simplemente enlazada. Entonces, de la información de configuración completada por el usuario, el lenguaje de programación y la plataforma utilizada son encontradas (C y Linux de 32-bit, por ejemplo). Del código fuente se obtiene el tipo de *valor* (`short` en este caso). Finalmente, los límites para el tipo `short` para el lenguaje de programación C y para la plataforma Linux de 32-bit pueden ser buscados en la base de datos y se aplica la táctica con estos valores.

Otro aspecto del análisis realizado para automatizar la aplicación de las tácticas de testing refiere a la selección del mejor nodo del árbol de testing desde el cual la táctica debe aplicarse. Si bien todas las tácticas podrían aplicarse sobre la raíz del árbol de clases para cada operación, en la mayoría de los casos se producirían clases vacías o clases que no mejorarían sustancialmente la cobertura obtenida. Entonces, Fastest implementa un análisis diferente para cada táctica, en pos de encontrar los nodos donde tiene sentido aplicar cada táctica y evitar la generación de clases no deseadas:

SP Si la expresión es parte de la precondición de la operación, entonces la táctica se aplica a todas las clases de prueba donde la expresión está presente. Cuando la expresión es parte de la postcondición, entonces la táctica se aplica a todas las clases de prueba, cuyos predicados son equivalentes a las precondiciones que llevan a la postcondición (esto se resuelve mediante el uso de DNF).

[2]Incluyendo información tales como la plataforma objetivo, el lenguaje de programación, etc.

FT Esta táctica se aplica a todas las clases de prueba donde la variable considerada se encuentra presente.

NR Esta táctica se aplica a todas las clases de prueba donde la expresión considerada se encuentra presente.

ISE Esta táctica se aplica a todas las clases de prueba donde alguno de $expr, epxr_1, ..., expr_n$ está presente.

Finalmente, cada estrategia de testing llama automáticamente a `prunett` luego de que cada táctica ha sido aplicada. De hecho, la presencia de casos de prueba insatisfacibles es otra de la razones por las cuales, estrategias de testing más fuertes no producen necesariamente más casos de prueba que las más débiles.

Capítulo 5

Lenguaje de Estrategias

En el capítulo anterior introdujimos la definición del concepto de estrategia de testing y definimos algunas de ellas, de forma que el usuario sin conocimientos del TTF o de la especificación en particular las puede utilizar. A continuación definiremos el Fastest Testing Strategies Definition Language (FTSDL), un lenguaje de scripting que permite definir tanto las estrategias vistas como otras nuevas, y el cual permitirá a los usuarios avanzados tener un mayor control sobre la manera en la que se forman las clases de prueba. Este lenguaje es de alto nivel y no requiere conocimientos de Java, por lo que resulta simple de usar.

5.1. Construyendo una Estrategia

Introduciremos FTSDL a medida que definimos una de las estrategias vistas en el capítulo anterior utilizando el lenguaje. Pero para eso empezaremos por definir una estrategia más simple, la cual iremos modificando. La llamaremos SPCompleta, ya que aplica la táctica SP toda las veces que puede sobre todos los nodos del árbol:

```
1 STRATEGY SPCompleta
2     FOREACH( e : getSPExpressions(OP) )
3         addtactic OP SP e.op e
4     END
```

```
5     genalltt
6     prunett
7 END
```

La primera línea indica tanto el comienzo de la definición de una estrategia como el nombre de la misma, en este caso, SPCompleta. El indicador `STRATEGY`, seguido del nombre que deseamos asignarle, determina el inicio de la definición de la estrategia, la cual realizará el trabajo comprendido entre esta línea y la palabra `END` correspondiente (en este ejemplo en la línea 7). El usuario puede separar la definición de varias estrategias dentro de un mismo archivo indicando el inicio de cada una con la sentencia `STRATEGY` y delimitando su contenido con la palabra clave `END`. Luego, puede cargar el archivo con las estrategias en Fastest y utilizar cualquiera de ellas para generar las clases de prueba de una determinada operación.

En el cuerpo de la estrategia definimos el comportamiento que esperamos para la misma, mediante la utilización de distintas *sentencias*. En nuestro ejemplo, la primera de ellas es una sentencia `FOREACH`, la cual nos permite definir bucles en el lenguaje. Este tipo de sentencias consta de dos partes. La primera, entre paréntesis, define una variable, en este caso `e`, la cual tomará distintos valores del conjunto `getSPExpressions(OP)` en cada ciclo del bucle. La segunda parte indica lo que se realizará cada vez que `e` tome uno de esos valores. Por ahora, simplemente aplicará una táctica.

`getSPExpressions` es un operador del lenguaje que permite obtener las expresiones con las que se puede aplicar la táctica SP sobre una determinada operación. Este operador tiene como parámetro el nombre de una operación (`OP` en este ejemplo) y retorna el conjunto de expresiones que pertenecen a esa operación y utilizan un operador de Z para el cual hay una partición estándar definida. De esta manera, el bucle realizará una iteración para cada uno de estos elementos. `OP` es un indicador provisto por el lenguaje que hace referencia al esquema sobre el que se aplica la estrategia, por ejemplo, si la estrategia se aplica sobre el esquema *Reservar*, `OP` tomará este valor durante la ejecución y `getSPExpressions(OP)` obtendrá las expresiones disponibles para aplicar SP encontradas en ese esquema.

La sentencia `addtactic OP SP e.op e` indica a Fastest que debe seleccionar la táctica `SP` sobre la operación `OP`, utilizando la expresión `e` y su operador. La variable `e` cambiará su valor en cada iteración, pasando por todas las expresiones sobre las que es posible aplicar SP.

Entonces, la estrategia primero buscará las posibles expresiones (y el operador utilizado) para aplicar SP sobre una determinada operación. Luego, por cada uno de estos valores, ejecutará el comando `addtactic`.

En las líneas 5 y 6 se introducen dos nuevas sentencias, las cuales se ejecutarán una vez que el bucle haya terminado. `genalltt` y `prunett` indican a Fastest que debe generar el árbol de clases con las tácticas seleccionadas previamente y podar los nodos vacíos, respectivamente, dando por terminada la estrategia.

La estrategia recién definida es una versión aproximada a la estrategia MATEMATICA vista en el capítulo anterior. La diferencia principal es que en SPCompleta se aplica SP sobre todos los nodos, de hecho, primero se seleccionan todas las tácticas a aplicar y recién al final se genera el árbol utilizándolas. De esta manera, el comando `genalltt` aplicará DNF sobre el VIS de la operación y sobre todos los nodos DNF generados se aplicará cada táctica, produciendo entonces una gran cantidad de nodos vacíos y cálculos innecesarios.

La idea entonces es modificar esta estrategia para que se comporte como MATEMATICA, la cual, aplica SP con una expresión sólo sobre los nodos donde está presente (o la precondición que lleva a la expresión está presente). Para ello, incluimos en el lenguaje una manera de determinar si una expresión `e` se encuentra (o si la precondición necesaria lo hace) en un determinado nodo `n`:

```
e IN n
```

Entonces podemos aplicar la táctica sólo en los casos donde se verifique esta condición. Para esto, utilizamos la sentencia `IF`:

```
IF ( e IN n )
    addtactic n SP e.op e
END
```

La sentencia `IF` ejecutará entonces el comando `addtactic` solo cuando la expresión `e IN n` se verifique, y no realizará nada cuando no.

Recordemos que la estrategia MATEMATICA aplica primero DNF y luego SP todas las veces que puede (o debe), sobre los nodos generados por DNF. Entonces, en nuestra implementación de esta estrategia necesitamos una manera de obtener los nodos generados por DNF. Para eso, utilizamos la expresión:

```
treeRoot(OP).leaves()
```

La primera expresión, `treeRoot(OP)`, determina la raíz del árbol de clases de la operación, y la segunda expresión obtiene las hojas del árbol. De esta manera, si al comienzo de la operación incluimos la sentencia `genalltt`, Fastest antes que nada generará el árbol para la operación aplicando DNF y creando así las primeras hojas. Luego la expresión `treeRoot(OP).leaves()` tendrá como valor el conjunto de hojas generadas por DNF. Como hicimos anteriormente, podemos iterar sobre este conjunto con una sentencia `FOREACH`, donde la variable `n` representará cada uno de estos nodos:

```
FOREACH( n : treeRoot(OP).leaves() )
    IF ( e IN n )
        addtactic n SP e.op e
    END
END
```

Podemos utilizar los identificadores predefinidos `TI` y `APPLIED` para determinar cuantas tácticas de testing debemos aplicar. `TI` indica la intensidad de testing que el usuario eligió al ejecutar la estrategia, mientras que `APPLIED` lleva la cuenta de la cantidad de tácticas que se han aplicado desde el comienzo de la ejecución. En ese momento, el intérprete inicializa su valor en 0, y cada vez que una táctica es aplicada, aumenta su valor. Entonces podemos hacer uso de estos identificadores para verificar si hemos alcanzado la cantidad suficiente de tácticas aplicadas. Si esto ocurre, damos por terminada la ejecución mediante la sentencia `exit` luego de usar los comandos `genalltt` y `prunett`:

```
IF (APPLIED == TI)
    genalltt
    prunett
    exit
END
```

Nos resta entonces encontrar las expresiones para la táctica como hicimos anteriormente y obtenemos la estrategia MATEMATICA:

```
STRATEGY Matematica
    genalltt
    FOREACH( e : getSPExpressions(OP) )
        FOREACH( n : treeRoot(OP).leaves() )
            IF( e IN n )
                addtactic n SP e.op e
                IF(APPLIED == TI)
                    genalltt
                    prunett
                    exit
                END
            END
        END
    END
    genalltt
    prunett
END
```

Se puede notar que podríamos invertir el orden de las sentencias `FOREACH`, iterando primero sobre las hojas del árbol y buscando las expresiones SP para cada una de ellas. El problema con esta forma de hacerlo es que si buscamos la expresiones utilizando la expresión `getSPExpressions(n)`, solo encontraremos expresiones definidas en las hojas, y no aquellas que formen parte de la postcondición de la operación.

La estrategia MATEMATICA Y LIMITES puede ser definida utilizando como base la estrategia recién definida, lo único que falta es aplicar la táctica NR

cuando se pueda de la misma forma que se hizo con SP. En vez de utilizar `getSPExpressions`, deberemos utilizar `getNRExpressions` como podría imaginarse y podemos iterar sobre este conjunto como hicimos anteriormente. Un punto a notar es que ambas expresiones devuelven conjuntos, pero, internamente, los tipos de estos conjuntos son distintos en el intérprete. En el primer caso, los elementos del conjunto son de tipo *SPExpr*, mientras que en el segundo tienen el tipo *NRExpr*. No hay grandes diferencias en estas estructuras, simplemente podemos realizar distintas operaciones sobre ellas, como obtener el operador involucrado (`e.op`) si es de tipo *SPExpr* o el rango de valores a utilizar (`e.ran`) para *NRExpr*.

Cuando no podamos determinar automáticamente el tipo en el lenguaje de implementación de la variable o expresión, debido a la falta de documentos de refinamiento, cuando se aplique la táctica NR, Fastest preguntará al usuario cuál es el tipo con el que fue implementada la expresión, para concluir de esta manera los valores con los que debe generar las clases.

Otra diferencia que podemos marcar entre las estructuras internas como *SPExpr* y *NRExpr* es el comportamiento que tendrá la expresión `e IN n`. Como vimos en el Capítulo 4, cada táctica debe aplicarse sobre un nodo `n` en distintas situaciones. Entonces, cuando la expresión `e` sea de tipo *NRExpr*, el operador `IN` determinará si la expresión se encuentra presente en el nodo `n`, mientras que para *SPExpr*, deberá indicar si la expresión es parte de la precondición o de la postcondición de la operación, y en base a dónde se encuentre, señalar si el nodo `n` debe ser objetivo de la táctica. Para los casos donde se busquen expresiones para las tácticas ISE y FT, se generarán expresiones de tipo *ISEExpr* y *FTExpr* respectivamente, las cuales se comportan de forma similar a las ya mencionadas.

En el Apéndice B se pueden ver las definiciones de las estrategias propuestas en este trabajo. Allí, las estrategias hacen uso de la invocación de otras estrategias en su definición, por lo tanto, la definición de la estrategia MATEMATICA recién vista difiere levemente de la presente en el apéndice, aunque su comportamiento es el mismo.

5.2. Gramática y Semántica del Lenguaje

En esta sección del capítulo veremos la gramática de las construcciones del lenguaje de estrategias, así como su semántica. Para definir la gramática utilizaremos la notación de Backus-Naur (BNF). La gramática completa puede verse en el Apéndice A. Explicaremos la semántica en base al código Java generado, ya que es con este lenguaje con el que se implementó tanto Fastest como el intérprete de FTSDL. El intérprete toma como entrada código escrito en este lenguaje, y ejecuta una serie de comandos Fastest (`genalltt`, `prunett` ó `addtactic`) como salida, a medida que computa los argumentos de estos comandos.

Las dos primeras reglas en la definición de la gramática son *strategy_document* y *strategy*:

⟨*strategy_document*⟩ ::= {⟨*strategy*⟩ {eol}}

⟨*strategy*⟩ ::= 'STRATREGY' ⟨*identifier*⟩ eol
{⟨*statement*⟩}
'END'

strategy_document es la regla de inicio del parser y básicamente indica que la entrada del intérprete puede contener la definición de varias estrategias de testing. Esto se indica encerrando la expresión entre llaves ({ y }) como muestra la gramática, lo que determina que la regla puede aparecer 0 o más veces. Esta regla se utiliza porque en Fastest se utilizará el comando `loadstrategy document`, donde `document` puede ser cualquier archivo que contenga la definición de estrategias en este lenguaje. Entonces, se puede concentrar la definición de muchas estrategias en el mismo documento:

```
STRATEGY A
    ...
    cuerpo de la estrategia A
    ...
END
```

```
STRATEGY B
    ...
    cuerpo de la estrategia B
    ...
END
```

Cada estrategia es parseada bajo la regla *strategy*, y determina que debe comenzar por la palabra `STRATREGY` seguida del nombre y *fin de línea* (*eol*), luego el contenido de la estrategia, y finalmente la palabra `END` para indicar que se termina su definición. El nombre de la estrategia es un identificador, que se utiliza en Fastest para determinar qué estrategia aplicar cuando se han definido muchas de ellas. El comando de Fastest:

```
applystrategy name operation intensity
```

aplicará entonces la estrategia cuyo nombre sea `name` sobre el esquema de operación o nodo `operation` con una intensidad de testing `intensity`.

El cuerpo de la estrategia se define mediante la utilización de uno o varios *statement* separados por *eol*. La regla correspondiente en la gramática es la siguiente:

⟨*statement*⟩ ::= ⟨*expression*⟩ eol
| ⟨*variable_assignment*⟩ eol
| '`exit`' eol
| ⟨*if_statement*⟩
| ⟨*while_statement*⟩
| ⟨*for_statement*⟩
| ⟨*foreach_statement*⟩

Un *statement* puede ser una sentencia de control de flujo, como *if_statement*, *for_statement*, *while_statement* o *foreach_statement*:

⟨*if_statement*⟩ ::= '`IF`' '`(`' ⟨*expression*⟩ '`)`' [eol] {⟨*statement*⟩}
['`ELSE`' [eol] {⟨*statement*⟩}] '`END`' eol

⟨*for_statement*⟩ ::= '`FOR`' '`(`' (⟨*variable_assignment*⟩ | [⟨*expression*⟩]) '`;`'
[⟨*expression*⟩] '`;`' [⟨*expression*⟩] '`)`' [eol] {⟨*statement*⟩}
'`END`' eol

⟨*while_statement*⟩ ::= '`WHILE`' '`(`' ⟨*expression*⟩ '`)`' [eol]
{⟨*statement*⟩} '`END`' eol

⟨*foreach_statement*⟩ ::= '`FOREACH`' '`(`' ⟨*identifier*⟩ '`:`' ⟨*expression*⟩ '`)`' [eol]
{⟨*statement*⟩} '`END`' eol

Estas sentencias son utilizadas como en la mayoría de los lenguajes de programación imperativos, tanto para verificar si se cumplen ciertas condiciones durante la ejecución del código, como para realizar bucles, es decir, ejecutar varias veces distintas líneas de código. Estas sentencias serán leídas por el intérprete y ejecutadas utilizando código Java, con la misma semántica en ambos lenguajes. Básicamente se traducirá de la gramática utilizada aquí para generar un bucle o un condicional en Java según corresponda. Es decir, cuando el intérprete lea una sentencia *if_statement*, generará en Java una sentencia `if`, donde la condición a cumplir será el resultado de traducir la *expression* en la definición de la regla, y como cuerpo del condicional se incluirá el resultado de traducir las sentencias *statement*. De la misma manera se traducirán las sentencias *while_statement* a un `while` en Java y tanto *for_statement* como *foreach_statement* traducen a un `for` en Java. Para la sentencia en *foreach_statement* en particular, se utilizará una forma especial de `for`, disponible a partir de Java 1.5, la cual permite definir un bucle para recorrer una estructura iterable de forma muy simple.

Una de las sentencias utilizadas comúnmente es *variable_assignment*, la cual nos permite definir una variable de nuestro lenguaje y asignarle un valor. Luego podemos acceder (y modificar) este valor utilizando el nombre de la variable, como en todos los lenguajes de programación imperativos:

⟨*variable_assignment*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*expression*⟩

Podemos por ejemplo, definir una variable `tactica`, la cual podríamos utilizar en una estrategia para determinar la táctica a aplicar. Y podemos asignarle el valor `SP`, que representa la táctica particiones estándar:

```
tactic = SP
```

de esta manera, podremos luego utilizar esta variable en representación de una táctica como argumento para el comando `addtactic`, o podremos cambiar su valor, por ejemplo, dentro de una sentencia `if`. En este lenguaje no es necesario definir el tipo de la variable, y podrá ser utilizada en distintas situaciones con expresiones de diferente tipo. El intérprete creará en Java una variable con el mismo nombre, utilizando una clase definida para ello, la cual internamente puede almacenar valores de varios tipos (String, Integer, Node, etc). A su vez, cada vez que un valor sea asignado a la variable, se determinará el tipo de la expresión y se almacenará debidamente en la variable. De forma que, cuando se necesite usar el valor de la variable, se puede utilizar el método correspondiente de la clase para obtener el valor del tipo correspondiente. Entonces permitimos que el usuario defina y utilice variables sin pensar en el tipo correspondiente con la que se definirá en Java.

Las reglas vistas hacen constante uso de *expression*, regla que se utiliza para calcular valores, llamar a funciones o invocar comandos de Fastest:

⟨*expression*⟩ ::= ⟨*numeric_expression*⟩
| ⟨*testing_expression*⟩
| ⟨*logical_expression*⟩
| ⟨*literal_expression*⟩
| ⟨*tactic*⟩
| ⟨*z_operator*⟩
| ⟨*identifier*⟩
| '`(`' ⟨*expression*⟩ '`)`'
| ⟨*command*⟩
| ⟨*function*⟩ '`(`' [⟨*arglist*⟩] '`)`'
| ⟨*expression*⟩ '`(`' [⟨*arglist*⟩] '`)`'
| ⟨*expression*⟩ '`.`' ⟨*expression*⟩
| '`OP`'
| '`TI`'
| '`APPLIED`'

Como podemos ver, *expression* representa una gran cantidad de construcciones, entre las cuales encontramos *command*. Esta es una de las reglas más importantes, ya que es la que se utiliza para invocar comandos de Fastest y por ende, define el comportamiento de la estrategia:

⟨*command*⟩ ::= '`genalltt`'
| '`prunett`'
| '`addtactic`' ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩ [⟨*expression*⟩]
| '`applystrategy`' ⟨*expression*⟩ ⟨*expression*⟩
| '`exit`' [⟨*expression*⟩]

El intérprete traducirá estos comandos a Java, como una llamada a las funciones de Fastest que definen el comportamiento de los comandos. Tanto `genalltt` como `prunett`, no utilizan argumentos, por lo tanto se traducen directamente a las funciones correspondientes. En el caso de `addtactic`, es necesario primero determinar los parámetros. La gramática establece que deben ser cuatro expresiones (con algunas tácticas 3), las cuales se utilizarán para determinar el nodo u operación sobre la que se aplicará la táctica, qué táctica aplicar, y luego los dos (o uno) argumentos específicos de la táctica (como ser el operador y la expresión donde aparece si utilizamos SP, por ejemplo). Entonces, el intérprete traducirá primero estos parámetros y luego invocará la función de Fastest con las traducciones correspondientes. El caso de `applystrategy` se utiliza de la misma manera, con la diferencia de que en vez de aplicar un táctica, aplicará un estrategia completa.

El comando `exit` no es un comando propio de Fastest, pero fue añadido en el lenguaje para poder dar por finalizada una estrategia cuando así se desee. En los ejemplos vistos anteriormente fue utilizado para dar por terminada una estrategia cuando se alcanza la intensidad de testing. La aplicación de una estrategia se da por terminada tanto cuando se encuentra la ejecución del comando `exit` como cuando se termina de recorrer el código que la define. En ambos casos, la estrategia retorna como valor la cantidad de tácticas que se aplicaron durante su ejecución. Para ello, durante la ejecución de una estrategia, el intérprete lleva la cuenta de la cantidad de aplicaciones de tácticas realizas. Este valor puede ser accedido en FTSDL utilizando la

expresión `APPLIED`.

expression además puede ser alguna de las siguientes construcciones:

- *numeric_expression*: son construcciones numéricas, que nos permiten utilizar operadores como +, ++, ∗ entre otros. Expresiones de este tipo se comportan de la misma manera en este lenguaje como en Java. El intérprete las traducirá a este último utilizando los mismos operadores, ya que todos ellos se encuentran definidos. Estos operadores, pueden ser aplicados a cualquier expresión, ya sean variables, el resultado de funciones u otras expresiones, teniendo en cuenta que siempre deben ser numéricas. De no ser así, cuando el intérprete realiza la traducción a Java, intentará aplicar operadores matemáticos sobre expresiones que no tienen el tipo correspondiente, resultando en un error. La gramática para esta regla (y para cualquier otra) se puede ver en el apéndice A.

- *testing_expression*: estas expresiones son utilizadas mayormente en construcciones como `if`, `for` y `while`, ya que permiten comparar expresiones, dando como resultado valores booleanos, y utilizar estos como condicionales en las construcciones mencionadas. Esta regla define operadores para ser utilizados entre expresiones como $<$, $\geq$, $\neq$ entre otras similares. El intérprete traducirá estas expresiones a Java utilizando los mismo operadores en ambos lenguajes, tal como se describe para *numeric_expression*. La única excepción es cuando se utiliza el operador `IN`. En este caso el intérprete deberá invocar una función en Java la cual determina si una expresión se encuentra en un nodo como explicamos anteriormente. Esta función es definida como parte de Fastest, y se comporta de manera diferente en base al tipo de expresión que recibe como argumento. Por ejemplo, si buscamos las expresiones a utilizar en la táctica NR, utilizamos `getNRExpressions`, donde cada elemento tendrá el tipo NRExp en Java. Entonces, si tomamos uno de estos elementos como parte de la sentencia `IN`, el intérprete determinará que deseamos ver si la expresión se encuentra en un determinado nodo y que deseamos corroborar esto porque aplicaremos la táctica NR. De esta manera, se invoca el método correspondiente para esta táctica.

- *logical_expression*: de forma similar a las anteriores reglas, se definen aquí operadores lógicos como $\wedge$ (&&) y $\vee$ (||). Estos son traducidos de la misma manera que los anteriores, es decir, de forma directa a Java.

- *tactic*: hace referencia a las tácticas que podemos usar para definir nuestra estrategia. Actualmente podemos utilizar SP, NR, ISE o FT. DNF se aplicará sólo cuando utilicemos el comando `genalltt`. Expresiones de este tipo son manejadas como String en Java.

- *z_operator*: estos son los operadores de Z, los cuales son utilizados como argumento para la táctica SP. Al igual que las tácticas, se manejan como String en Java.

- *identifier*: representa una variable. Esta expresión puede ser utilizada tanto para cuando queremos asignarle un valor a la variable, como cuando querramos utilizar el valor que fue almacenado en la misma. Como vimos anteriormente, en el primer caso, en Java se creará una variable con el mismo nombre, si aún no fue creada, y se almacenará su valor. En el segundo caso, se deberá determinar el contexto donde es utilizada la expresión para determinar así su tipo y obtener el valor correspondiente en Java.

- *function*: definimos una variedad de funciones o métodos los cuales pueden ser utilizados para definir las estrategias. Algunos de estas funciones se utilizan para acceder a cierta información de Fastest, como ser el árbol de clases de prueba, o información que debe ser obtenida de la especificación, como los posibles argumentos de una táctica. En cualquiera de estos casos, el intérprete simplemente deberá invocar a los métodos correspondientes en Java, algunos de los cuales hemos definido nosotros y otros son simplemente métodos de los tipos utilizados ya provistos. Como parte de este proceso, deberá primero traducir las expresiones que se utilizan como argumentos. Por ejemplo, si el usuario utiliza `node.isLeaf`, el intérprete deberá llamar al método `isLeaf` definido por nosotros, el cual utiliza el método `hasChildNodes` propio de la clase `Node`, para determinar si el elemento `node` es una hoja.

- `OP`: el usuario puede acceder al nombre de la operación a la cual se le aplica la táctica utilizando esta expresión. En Fastest, se utiliza el comando `applytactic` donde uno de los argumentos es el nombre de una operación de la especificación (o un nodo si el árbol ya fue creado). En Java, este nombre es almacenado en esta variable, la cual será utilizada cada vez que se necesite traducir esta expresión.

- `TI`: este identificador hace referencia a la intensidad de testing con la que se invocó la estrategia. Su valor puede ser accedido dentro de la estrategia, generalmente, para determinar si en un determinado momento se han aplicado la cantidad requerida de tácticas.

- `APPLIED`: análogamente a la intensidad de testing, este identificador permite ver la cantidad de tácticas que se aplicaron durante la ejecución de la estrategia.

El resto de las reglas en la gramática definen los detalles de la sintaxis, como las cadenas de caracteres aceptadas como identificadores para las variables, o los números que se pueden utilizar. Estas reglas pueden verse en el Apéndice A ya que no aportan ninguna complejidad al lenguaje ni a la manera en que debe traducirse el mismo.

Capítulo 6

Conclusión y Trabajos Futuros

El TTF no es el único método de MBT que trabaja con la notación Z [22] [23] [24] [25] [26] [27] [28]. Algunos de estos métodos también hacen uso de tácticas de testing o conceptos similares. Sin embargo, ninguno de ellos ofrece todas las tácticas de testing mostradas en esta tesina; no combinan las tácticas como lo hace el TTF; y tampoco ofrecen el nivel de automatización alcanzado por Fastest luego de la definición de las estrategias de testing. Por otro lado, algunos de estos métodos [24] [25] describen los casos de prueba como secuencias de operaciones que ejecuta la implementación. Esta es la forma en la que trabajan muchos métodos de MBT para otras notaciones. En estos casos, los usuarios pueden extraer las secuencias al recorrer un autómata. De hecho, pueden aplicar diferentes criterios de testing para recorrer el autómata de diferentes maneras, lo que genera diferentes conjuntos de secuencias. Este concepto es similar a las estrategias de testing propuestas en este trabajo, aunque creemos que las estrategias brindan una idea más cercana sobre cómo la implementación será testeada.

Los resultados presentados en este trabajo son otra indicación de que el TTF puede ser automatizado como cualquier otro método de MBT, tanto para la notación Z como para otros lenguajes [13] [29]. Antes de obtener estos resultados, Fastest proveía un nivel de automatización similar a otras herramientas de MBT en cuanto a la generación de casos de prueba, eliminación de condiciones de prueba insatisfacibles, traducción de casos de prueba

a lenguaje natural [30] y refinamiento y abstracción de casos de prueba [21]. Entonces, el concepto de estrategia de testing viene de notar que la aplicación de tácticas de testing aún requería cierto trabajo manual que podría no ser el adecuado para algunos grupos de usuarios. Además, las estrategias de testing propuestas provienen de la experiencia y conocimiento obtenidos luego de aplicar Fastest a varios proyectos y casos de estudio [31] [17]. En este sentido, nuestra idea de estrategia de testing no es sólo la unión de algunos comandos en forma de script, sino que realmente ayudan al usuario a evitar realizar un análisis no trivial mediante la implementación de algunas heurísticas de testing.

Las estrategias de testing sólo contribuyen a la automatización del proceso de testing basado en el TTF y Fastest. Lo que significa que no cambian la teoría subyacente ni las técnicas básicas. De hecho, los usuarios pueden combinar estrategias y tácticas para dar forma al árbol de testing como ellos deseen. Por ejemplo, luego de aplicar una estrategia de testing, los usuarios pueden aplicar más tácticas en diferentes subárboles del árbol de testing y podar manualmente aquellos nodos que no deseen por alguna razón. El orden parcial que las organiza puede ser extendido o modificado a medida que se crean nuevas tácticas o se modifican las existentes. De hecho, Fastest ofrece algunas tácticas que no han sido consideradas en esta tesina [32], algunas de las cuales pueden ser utilizadas en nuevas y más fuertes estrategias.

Aunque la Figura 4.2 debería ayudar a los usuarios a elegir la estrategia correcta para cada operación Z, cada estrategia de testing debe ser acompañada de una descripción listando sus características principales en términos del cubrimiento que ofrecen, estilo de los casos de prueba que generan o aspectos que ejercitan. El vocabulario de estas descripciones debe ser acorde al de la comunidad de testing, en vez de aquel de Z, el TTF o las comunidad de MBT. Más información técnica puede ser provista para usuarios avanzados, quienes además pueden definir o modificar estrategias en los casos donde se necesita generar casos de prueba de forma particular.

Entonces, la conclusión principal que obtenemos de este trabajo es que, con respecto a los métodos de MBT, la partición de dominios puede ser realizada de forma automática desde un nivel de abstracción bastante alto y

con un vocabulario más orientado al testing. Además, el orden parcial que puede ser definido entre las estrategias permite a los testers elegir la correcta a partir de lo que será testeado a niveles de implementación en vez de por cómo se particiona el dominio de entrada. Este orden parcial puede ser modificado y extendido con nuevas estrategias a medida que se definen y mejoran.

En el futuro planeamos terminar con la implementación total del lenguaje FTSDL; pensamos escribir tarjetas descriptivas que ayuden a los testers a entender de forma informal lo que cada estrategia va a testear y la relación que hay entre ellas; y deseamos explorar si algunas tácticas de testing que no han sido consideradas todavía, pueden ser utilizadas para definir nuevas estrategias de testing.

Bibliografía

[1] G. J. Myers, *The Art of Software Testing*. John Wiley and Sons, 1979.

[2] M. Utting y B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[3] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward y H. Zedan, *Using formal specifications to support testing*. ACM Comput. Surv., vol. 41, no.2, pp. 1-76, 2009.

[4] W. Grieskamp, N. Kicillof, K. Stobie y V. A. Braberman, *Model based quality assurance of protocol documentation: tools and methodology*. Softw. Test., Verif. Reliab., vol. 21, no.1, pp.55-71, 2011.

[5] M. S. A. Trab, M. Brockway, S. Counsell y R. M. Hierons, *Testing real-time embedded systems using timed automata based approaches*. Journal of Systems and Software, vol. 86, no. 5, pp. 1209-1223, 2003.

[6] J. Peleska, *Industrial-strength model-based testing - state of the art and current challenges*. CoRR, vol. abs/1303.1006, 2013.

[7] J. Zander, I. Schieferdecker y P. Mosterman, *Model-based Testing for Embedded Systems*. ser. Computational Analysis, Synthesis, and Design of Dynamic System Series. CRC Press, 2012.

[8] Stocks P., *Applying formal methods to software testing*. PhD Thesis, Department of Computer Science, University of Queensland, 1993.

[9] P. Stocks y D. Carrington, *A Framework for Specification-Based Testing*. IEEE Transactions on Software Engineering, vol. 22, no. 11, pp. 777-793, 1996,

[10] L. Murray, D. A. Carrington, *A Framework for Specification-Based Testing*. IEEE Computer Society, pp. 80-87, 1997,

[11] J. M. Spivey, *The Z notation: a reference manual*. Hertfordshire, UK, UK: Prentice Hall Inter-national (UK) Ltd., 1992.

[12] ISO. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*. Technical Report ISO/IEC 13568, International Organization for Standardization 2002.

[13] M. Cristiá, P. Albertengo, C. Frydman, B. Plüss y P. R. Monetti, *Tool support for the Test Template Framework*. Softw. Test., Verif. Reliab.

[14] M. Cristiá y P. Rodriguez Monetti, *Implementing and applying the Stocks-Carrington framework for model-based testing*. ICFEM, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., vol. 5885. 2009, pp. 167-185.

[15] M. Cristiá, P. Albertengo y P. Rodriguez Monetti, *Prunning testing trees in the Test Template Framework by detecting mathematical contradictions*. SEFM, J. L. Fiadeiro and S. Gnesi, Eds. IEE Computer Society 2010, pp. 268-277.

[16] M. Cristiá, P. Albertengo y P. Rodriguez Monetti, *Fastest: a model-based testing tool for the Z notation*. in PTD-SEFM, F. Mazzanti and G. Trentani, Eds. Consiglio Nazionale della Richerche, Pisa, Italy, 2010 pp. 3-8.

[17] M. Cristiá, P. Albertengo, C. S. Frydman, B. Plüss y P. Rodriguez Monetti, *Applying the Test Template Framework to aerospace software*. in SEW, J. L. Rash and C. Rouff, Eds. IEEE COmputer Society, 2011, pp. 128-137 pp. 3-8.

[18] M. Cristiá, G. Rossi and C. Frydman, {*log*} *as a test case generator for the test template framework*. Dipartimento de Matematica, Università di Parma, Tech. Rep., 2012.

[19] Cristiá M. *Fastest: Improving and Supporting the Test Template Framework*. Ph.D. dissertation, Faculté des Sciences et Techniques, Aix-Marseille Université, 2012.

[20] S Rapps, E.J. Weyuker. *Data flow analysis techniques for test data selection*. ICSE '82: Proceedings of the 6th international conference on Software engineering. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 272-278.

[21] M. Cristiá, D. Hollmann, P. Albertengo, C. S. Frydman, P.R. Monetti. *A language for test case refinement in the Test Template Framework*. ICFEM, ser. Lecture Notes in Computer Science, S. Qin y Z. Qiu, Eds., vol. 6991. Springer, 2011.

[22] P. Ammann y J. Offutt, *Using formal methods to derive test frames in category-partition testing*, in Compass'94: 9th Annual Conference on Computer Assurance. Gaithersburg, MD: National Institute od Standards and Technology, 1994, pp.69-80.

[23] P. A. V. Hall, *Towards testing with respect to formal specification*, in Proc. Second IEE/BCS Conference on Software Engineering, ser. Conference Publication, no. 290. IEE/BCS, Jul. 1988, pp.159-163.

[24] R. M. Hierons, S. Sadeghipour, y H. Singh, *Testing a system specified using Statecharts and Z*, Information and Software Technology, vol. 43, no. 2, pp.137-149, February 2001.

[25] R. M. Hierons, *Testing from a Z specification*, Software Testing, Verification & Reliability, vol. 7, pp.19-33, 1997.

[26] H. M. Hörcher y J. Peleska, *Using Formal Specifications to Support Software Testing*, Software Quality Journal, vol. 4, pp.309-327, 1995.

[27] S. Burton, *Automated Testing from Z Specifications*, Department of Computer Science - University of York, Tech. Rep., 2000

[28] S. Stepney, *Testing as abstraction*, in ZUM, ser. Lecture Notes in Computer Science, J. P. Bowen and M.G. Hinchey, Eds., vol. 967. Springer, 1995, pp.137-151.

[29] B. Legeard, F. Peureux, y M. Utting, *A Comparison of the BTT and TTF Test-Generation Methods*, in ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B. London, UK: Spring-Verlag, 2002, pp.309-329.

[30] M. Cristiá and B. Plüss, *Generating natural language descriptions of Z test cases*, in INLG, J. D. Kelleher, B. M. Namee, I. van der Sluis, A. Belz, A. Gatt, and A. Koller, Eds. The Association for Computer Linguistics, 2010, pp.173-177.

[31] M. Cristiá, V. Santiago, y N. Vijaykumar, *On comparing and complementing two MBT approaches*, in LATW, F. Vargas and E. Cota, Eds. IEEE Computer Society, 2010, pp.173-177.

[32] M. Cristiá y C. S. Frydman, *Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions*, in ABZ, ser. Lecture Notes in Computer Science, J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds., vol. 7316. Springer, 2012, pp.280-293.

Apéndice A

Gramática del lenguaje FTSDL

Lenguaje para la definición de Estrategias en Fastest

⟨*strategy_document*⟩ ::= {⟨*strategy*⟩ {eol}}

⟨*strategy*⟩ ::= 'STRATREGY' ⟨*identifier*⟩ eol
{⟨*statement*⟩}
'END'

⟨*statement*⟩ ::= ⟨*expression*⟩ eol
| ⟨*variable_assignment*⟩ eol
| ⟨*if_statement*⟩
| ⟨*while_statement*⟩
| ⟨*for_statement*⟩
| ⟨*foreach_statement*⟩

⟨*if_statement*⟩ ::= 'IF' '(' ⟨*expression*⟩ ')' [eol] {⟨*statement*⟩}
['ELSE' [eol] {⟨*statement*⟩}] 'END' eol

⟨*while_statement*⟩ ::= 'WHILE' '(' ⟨*expression*⟩ ')' [eol]
{⟨*statement*⟩} 'END' eol

⟨*for_statement*⟩ ::= ‘`FOR`’ ‘`(`’ (⟨*variable_assignment*⟩ | [⟨*expression*⟩]) ‘`;`’
[⟨*expression*⟩] ‘`;`’ [⟨*expression*⟩] ‘`)`’ [eol]
{⟨*statement*⟩} ‘`END`’ eol

⟨*foreach_statement*⟩ ::= ‘`FOREACH`’ ‘`(`’ ⟨*identifier*⟩ ‘`:`’ ⟨*expression*⟩ ‘`)`’ [eol]
{⟨*statement*⟩} ‘`END`’ eol

⟨*expression*⟩ ::= ⟨*numeric_expression*⟩
| ⟨*testing_expression*⟩
| ⟨*logical_expression*⟩
| ⟨*literal_expression*⟩
| ⟨*tactic*⟩
| ⟨*z_operator*⟩
| ⟨*identifier*⟩
| ‘`(`’ ⟨*expression*⟩ ‘`)`’
| ⟨*command*⟩
| ⟨*function*⟩ ‘`(`’ [⟨*arglist*⟩] ‘`)`’
| ⟨*expression*⟩ ‘`(`’ [⟨*arglist*⟩] ‘`)`’
| ⟨*expression*⟩ ‘`.`’ ⟨*expression*⟩
| ‘`OP`’
| ‘`TI`’
| ‘`APPLIED`’

⟨*numeric_expression*⟩ ::= ‘`-`’ ⟨*expression*⟩
| ‘`++`’ ⟨*expression*⟩
| ‘`--`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`++`’
| ⟨*expression*⟩ ‘`--`’
| ⟨*expression*⟩ ‘`+`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`-`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`*`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`/`’ ⟨*expression*⟩

⟨*testing_expression*⟩ ::= ⟨*expression*⟩ '`>`' ⟨*expression*⟩
| ⟨*expression*⟩ '`<`' ⟨*expression*⟩
| ⟨*expression*⟩ '`>=`' ⟨*expression*⟩
| ⟨*expression*⟩ '`<=`' ⟨*expression*⟩
| ⟨*expression*⟩ '`==`' ⟨*expression*⟩
| ⟨*expression*⟩ '`!=`' ⟨*expression*⟩
| ⟨*expression*⟩ IN ⟨*expression*⟩

⟨*logical_expression*⟩ ::= '`!`' ⟨*expression*⟩
| ⟨*expression*⟩ '`&&`' ⟨*expression*⟩
| ⟨*expression*⟩ '`||`' ⟨*expression*⟩
| '`true`'
| '`false`'

⟨*variable_assignment*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*expression*⟩

⟨*command*⟩ ::= '`genalltt`'
| '`prunett`'
| '`addtactic`' ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩
[⟨*expression*⟩]
| '`applystrategy`' ⟨*expression*⟩ ⟨*expression*⟩
| '`exit`' [⟨*expression*⟩]

⟨*function*⟩ ::= ⟨*tree_funct*⟩
| ⟨*parameters_funct*⟩
| ⟨*expression_funct*⟩
| ⟨*string_funct*⟩
| ⟨*enumeration_funct*⟩
| ⟨*tactic_funct*⟩

⟨*tree_funct*⟩ ::= '`treeRoot`'
| '`leaves`'

| 'getName'
| 'children'
| 'getParent'
| 'isLeaf'

⟨*parameters_funct*⟩ ::= 'getSPOperators'
| 'getSPExpressions'
| 'getNRExpressions'
| 'getFTVariables'
| 'getISEExpressions'

⟨*tactic_funct*⟩ ::= 'expressionSPAppearsIn'
| 'expressionNRAppearsIn'
| 'variableFTAppearsIn'
| 'expressionISEAppearsIn'

⟨*expression_funct*⟩ ::= 'getOperator'
| 'getString'

⟨*string_funct*⟩ ::= 'compare'

⟨*enumeration_funct*⟩ ::= 'hasMore'
| 'next'

⟨*tactic*⟩ ::= 'SP'
| 'NR'
| 'FT'
| 'ISE'

⟨*z_operator*⟩ ::= '\cup'
| '\cap'

| ‘\setminus’
| ‘<’
| ‘\leq’
| ‘>’
| ‘\geq’
| ‘=’
| ‘\neq’
| ‘\oplus’
| ‘\dres’
| ‘\ndres’
| ‘\rres’
| ‘+’
| ‘\in’
| ‘\notin’
| ‘\#’
| ‘\extract’

⟨*arglist*⟩ ::= ⟨*expression*⟩ { ‘,’ ⟨*expression*⟩ }

⟨*literal_expression*⟩ ::= ⟨*integer*⟩
| ⟨*string*⟩
| ⟨*char*⟩

⟨*identifier*⟩ ::= (‘a’..‘z’|‘A’..‘Z’) { (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’) }

⟨*integer*⟩ ::= (‘0’..‘9’) { (‘0’..‘9’) }

⟨*string*⟩ ::= ‘"’ { ⟨*char*⟩ } ‘"’

⟨*char*⟩ ::= cualquier caracter válido

Apéndice B

Estrategias de testing

Antes de poder definir las estrategias de testing presentadas anteriormente y utilizadas en Fastest, debemos definir cuatro estrategias auxiliares. Estas estrategias se centran únicamente en la selección de las diferentes tácticas utilizadas, a fin de facilitar luego la definición de las estrategias propuestas.

```
STRATEGY FTOnly
  FOREACH(v:getFTVariables(OP))
    addtactic OP FT v
    IF(APPLIED == TI)
      exit
    END
  END
END

STRATEGY SPOnly
  FOREACH(e:getSPExpressions(OP))
    FOREACH(n:treeRoot(OP).leaves())
      IF(e IN n)
        addtactic n SP e.op e
        IF(APPLIED == TI)
          exit
        END
```

```
      END
    END
  END
END

STRATEGY NROnly
  FOREACH(e:getNRExpressions(OP))
    FOREACH(n:treeRoot(OP).leaves())
      IF(e IN n)
        addtactic n NR e e.ran
        IF(APPLIED == TI)
          exit
        END
      END
    END
  END
END

STRATEGY ISEOnly
  FOREACH(e:getISExpressions(OP))
    FOREACH(n:treeRoot(OP).leaves())
      IF(e IN n)
        addtactic n ISE e
        IF(APPLIED == TI)
          exit
        END
      END
    END
  END
END
```

Ahora podemos definir las tácticas presentadas en el trabajo utilizando las definiciones auxiliares anteriores.

```
STRATEGY FuncionalidadBasica
  genalltt
  prunett
END

STRATEGY Enumeraciones
  genalltt
  applystrategy FTOnly TI
  genalltt
  prunett
END

STRATEGY LimitesDeImplementacion
  genalltt
  applystrategy NROnly TI
  genalltt
  prunett
END

STRATEGY ValoresImportantes
  genalltt
  applystrategy ISEOnly TI
  genalltt
  prunett
END

STRATEGY Matematica
  genalltt
  applystrategy SPOnly TI
  genalltt
```

```
  prunett
END

STRATEGY TodaFuncionalidad
  genalltt
  applystrategy SPOnly TI
  IF(TI == APPLIED)
    applystrategy FTOnly TI
  ELSE
    applystrategy FTOnly (TI-APPLIED)
  END
  genalltt
  prunett
END

STRATEGY MatematicaYLimites
  genalltt
  applystrategy SPOnly TI
  IF(TI == APPLIED)
    applystrategy NROnly TI
  ELSE
    applystrategy NROnly (TI-APPLIED)
  END
  genalltt
  prunett
END

STRATEGY BuenTesting
  genalltt
  applystrategy SPOnly TI
  IF(TI == APPLIED)
    applystrategy NROnly TI
  ELSE
```

```
    applystrategy NROnly (TI-APPLIED)
  END
  IF(TI == 2*APPLIED)
    applystrategy ISEOnly TI
  ELSE IF(TI > APPLIED)
    applystrategy ISEOnly (TI-APPLIED)
  END
  genalltt
  prunett
END

STRATEGY TestingFuerte
  genalltt
  applystrategy SPOnly TI
  IF(TI == APPLIED)
    applystrategy NROnly TI
  ELSE
    applystrategy NROnly (TI-APPLIED)
  END
  IF(TI == 2*APPLIED)
    applystrategy FTOnly TI
  ELSE IF(TI > APPLIED)
    applystrategy FTOnly (TI-APPLIED)
  END
  genalltt
  prunett
END

STRATEGY TestingCompleto
  genalltt
  applystrategy SPOnly TI
  IF(TI == APPLIED)
    applystrategy NROnly TI
```

```
  ELSE
    applystrategy NROnly (TI-APPLIED)
  END
  IF(TI == 2*APPLIED)
    applystrategy FTOnly TI
  ELSE IF(TI > APPLIED)
    applystrategy FTOnly (TI-APPLIED)
  END
  IF(TI == 3*APPLIED)
    applystrategy ISEOnly TI
  ELSE IF(TI > APPLIED)
    applystrategy ISEOnly (TI-APPLIED)
  END
  genalltt
  prunett
END
```