Poda de árboles de Testing a través de la detección de contradicciones matemáticas

Pablo Albertengo

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

Director: MSc. Maximiliano Cristiá

Marzo 2011

Resumen

Fastest es una herramienta que implementa y automatiza el *framework* para testing basado en modelos(**MBT**, por las siglas de Model Based Testing) para la notación Z creado por Phil Stocks y David Carrington. Dicho *framework* se denomina *Test Template Framework* (**TTF**).

En este trabajo se presenta una nueva funcionalidad de Fastest que ayuda al usuario del **TTF** a eliminar clases de test inconsistentes de manera automática.

El método es simple y práctico, y hace uso de ciertas peculiaridades del **TTF**.

Posiblemente las cualidades más importantes del método sean su extensibilidad y la sencillez que le brinda al usuario para su uso, ya que no asume conocimientos previos relativos a prueba de teoremas.

Índice general

Capítulo 1

Introducción

En [2] se presenta a Fastest [1], la primera herramienta automática que implementa parcialmente el *Test Template Framework* (**TTF**). El TTF es un *framework* para testing basado en modelos (**MBT**), especialmente adecuado para testing de unidad. Fue propuesto por Phil Stocks y David Carrington en [19] [18] [12].

En el TTF el espacio de entrada de una operación Z es particionado en lo que se denominan *clases de test*[2], las cuales dan lugar a los *árboles de test* como los que se observan en la Figura 2.2. Stocks y Carrington sugieren que los *casos de test* o *casos de prueba* deben ser derivados solo de las hojas de dichos árboles.

El problema que surge es que, debido a ciertas particularidades del TTF, pueden existir hojas de las cuales no es posible obtener un caso de prueba debido a que sus predicados son contradicciones o contienen ciertos términos indefinidos. Por lo tanto, estas hojas deberían ser podadas del árbol de test en una etapa previa al proceso de derivación de casos de prueba. El problema de la poda de árboles de testing se puede reducir entonces a determinar qué hojas del árbol son satisfactibles y descartar al resto, o bien, se puede reducir a identificar qué hojas no son satisfactibles y eliminarlas del árbol.

El primer camino implica el desarrollo de una herramienta del estilo de los SMT solvers[3] que implemente por completo el *toolkit* matemático de Z [15], lo cual, ciertamente, no es una tarea trivial dado el número de teorías implicadas en el toolkit. Precisamente, el rico lenguaje matemático empleado por los especificadores Z convierte al problema de la poda de árboles de test aún más complejo, ya que la mayoría de las contradicciones están relacionadas con contradicciones matemáticas y no con contradicciones lógicas.

En este trabajo tomaremos el segundo camino, es decir, trataremos de determinar qué clases no son satisfactibles. Nuestro enfoque está más relacionado con la prueba automática de teoremas, aunque no implicará la implementación de un probador de teoremas para la notación Z, ni la utilización de uno ya existente.

El problema de la poda de clases de prueba inconsistentes es tan importante como el problema de obtener casos de prueba a partir de las clases de prueba que son satisfactibles. Esto se debe principalmente a 2 causas:

El TTF tiende a generar muchas clases de prueba insatisfactibles

[1] Fastest está disponible públicamente en `http://www.flowgate.net` en la sección de Herramientas

[2] También denominadas objetivos de test, clases de prueba, etc.

[3] Un SMT solver se encarga de analizar la satisfactibilidad en cierto tipo específico de fórmulas lógicas de primer orden en las cuales ciertos predicados y símbolos de función tienen una interpretación basada en un conjunto de teorías asociadas de primer orden con igualdad

> Debido a que el problema de determinar el conjunto de clases de test que es satisfactible es "no decidible", cualquier método automático para derivar casos de prueba a partir de las clases de test producirá una solución incompleta. Entonces, si ese método no puede encontrar un caso de prueba para una clase determinada es porque la clase es insatisfactible o por el hecho de que el método es incompleto?

De cualquier manera, cualquier método automático para la poda de clases inconsistentes también será incompleto porque también es un problema "no decidible".

Precisamente por estos motivos una herramienta como Fastest debe implementar ambos métodos, ya que se complementan entre sí, dejando para inspección manual apenas un pequeño grupo de las clases de prueba.

En [2] se presenta un método simple pero práctico para encontrar casos de prueba a partir de clases de prueba (en promedio, el método puede encontrar un caso de prueba para el 90 % de las clases satisfactibles). En este trabajo introduciremos un método para la poda de clases de prueba con el objetivo de reducir dramáticamente el tiempo empleado para podar los árboles producidos por el TTF.

El trabajo se estructura de la siguiente manera. En el siguiente capítulo se da una breve introducción al TTF. El Capítulo 3 es el núcleo de este trabajo y es donde presentamos nuestro método. En el Capítulo 4 se brindan los detalles del método que utilizamos para certificar los teoremas de eliminación. En el Capítulo 5 se presentan algunos resultados que obtuvimos al probar nuestra implementación y en el siguiente Capítulo se compara con trabajos similares. Finalmente, en el Capítulo 7 se describen nuestras conclusiones y las líneas investigativas para futuros trabajos.

Capítulo 2

Test Template Framework

En este capítulo se introducirá brevemente el TTF por medio de un ejemplo, sin mencionar ninguna herramienta o implementación en particular. Se asume que el lector está familiarizado con la notación Z [17].

2.1. Un pequeño sistema de sensores

El modelo Z de la Figura 2.1 describe un pequeño sistema de sensores que almacena el valor máximo registrado por cada sensor.

La operación KMR[1] toma como entrada un ID de sensor, $s?$, y el valor de una lectura de dicho sensor, $r?$. Si $s?$ es un identificador válido y el valor de $r?$ es mayor que el valor almacenado para $s?$, KMR reemplaza el valor actual con $r?$. Si alguna de las condiciones no se cumple, la operación falla y no se produce ningún cambio.

2.2. Tácticas de testing, Arboles de Testing y Clases de Test

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

$$IS == [smax : SENSOR \pfun \mathbb{Z};\ s? : SENSOR;\ r? : \mathbb{Z}]$$

En tanto que el *VIS* es el subconjunto del *IS* para el cual la operación está definida. En el caso de *KMR* es igual a su *IS* pues la operación es total. Más formalmente, el *VIS* de una operación *Op* puede ser definido como sigue:

$$VIS_{Op} == [IS_{Op} \mid \text{pre } Op]$$

Stocks y Carrington sugieren dividir el *VIS* en clases de equivalencia, llamadas *clases de test* o *clases de prueba*, aplicando una o más *tácticas de testing*. Las clases de prueba obtenidas de esta manera pueden a la vez ser subdivididas en más clases de prueba aplicando otras tácticas

[1] *KMR* proviene de *KeepMaxReadings*.

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

KMROk
$\Delta MaxReadings$
$s? : SENSOR;\ r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$smax\ s? < r?$
$smax' = smax \oplus \{s? \mapsto r?\}$

KMRE2
$\Xi MaxReadings$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$

$KMRE1 ==$
$\quad [\Xi MaxReadings;\ s? : SENSOR \mid s? \notin \operatorname{dom} smax]$
$KMR == KMROk \lor KMRE1 \lor KMRE2$

Figura 2.1: Modelo Z para un pequeño sistema de sensores

de testing. Este proceso continúa hasta que el ingeniero se siente satisfecho con la cobertura de test obtenida. En el marco del TTF estas clases de prueba se representan en un *árbol de testing*, como el que se muestra en la figura 2.2. Los casos de prueba serán derivados solo de las hojas del árbol de testing.

En el TTF los autores definen un conjunto de tácticas de testing, aunque proponen la inclusión de nuevas tácticas cuando resulte necesario para distintos proyectos, sistemas, requerimientos, etc.

En este ejemplo aplicaremos 2 tácticas de testing a *KMR*. En primer lugar aplicamos Forma normal Disjuntiva (DNF) al *VIS* y en segundo lugar aplicamos la táctica de Particiones Estándar (SP) a la expresión $smax\ s? < r?$ de *KMROk* (ambas tácticas están explicadas en [19]). El árbol resultante se muestra en la Figura 2.2. Algunos de los nodos del árbol se muestran en la figura 2.3 como esquemas Z. Decidimos incluir en la figura algunas clases insatisfactibles porque son el principal tema de este trabajo.

El segundo paso en la metodología TTF sugiere la poda de las clases insatisfactibles del árbol de testing, porque es imposible derivar casos de prueba a partir de ellas. Por ejemplo, las clases *KMR_SP_9* y *KMR_SP_14* deben ser podadas – ver Figura 2.3. El predicado de *KMR_SP_9* es insatisfactible porque la expresión $smax\ s?$ está indefinida ya que $s? \notin \operatorname{dom} smax$, mientras que *KMR_SP_14* es inconsistente porque su predicado es una contradicción aritmética.

Stocks y Carrington no brindan ninguna sugerencia de cómo podría automatizarse la poda en el TTF. Precisamente el núcleo de este trabajo es brindar un método para automatizar dicho proceso y hasta donde hemos podido indagar es la primera propuesta en este sentido para el TTF, aunque pensamos que podría ser interesante para otros dominios también.

Luego de este paso, el TTF continúa derivando al menos un caso de prueba para cada una de las clases que sobrevivieron a la poda. Este paso se explica detalladamente en [18] y [2].

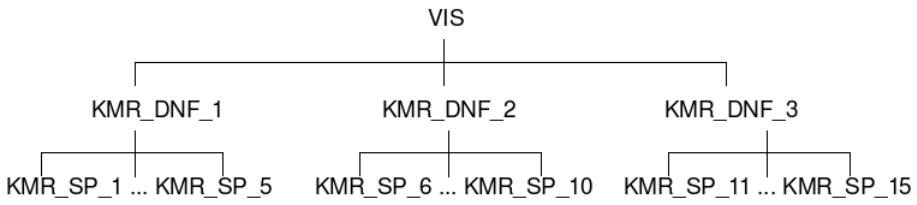


Figura 2.2: Arbol de testing de la operación *KMR*.

KMR_SP_9

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

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

SCAddCat_SP_1

$$level : \mathbb{Z}$$
$$categs : \mathbb{P}\, CATEGORY$$
$$c? : CATEGORY$$

$$c? \notin categs$$
$$categs = \{\}$$
$$\{c?\} = \{\}$$

Del_SP_1

$$procs : PID \pfun PSTATUS$$
$$p? : PID$$

$$p? \in \operatorname{dom}(procs \rhd \{waiting\})$$
$$procs = \{\}$$

KMR_SP_14

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

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

SCAddCat_SP_5

$$level : \mathbb{Z}$$
$$categs : \mathbb{P}\, CATEGORY$$
$$c? : CATEGORY$$

$$c? \notin categs$$
$$categs \neq \{\}$$
$$\{c?\} \neq \{\}$$
$$categs \subset \{c?\}$$

Deposit_SP_18

$$sa : NSA \pfun BALANCE$$
$$n? : NSA$$
$$a? : \mathbb{Z}$$

$$a? \leq 0$$
$$sa = \{\}$$
$$\{n? \mapsto (san? + a?)\} \neq \{\}$$

Figura 2.3: Esquemas Z que representan clases de prueba insatisfactibles del ejemplo de *KMR* y de otros ejemplos

Capítulo 3

Detectando Contradicciones Matemáticas

Fastest aplica por default la táctica DNF, convirtiendo así a los predicados de las clases de prueba en conjunciones de predicados atómicos. Las otras tácticas implementadas en Fastest no hacen más que conjugar nuevos predicados atómicos a los predicados ya existentes, tal como se prescribe en el TTF. Por lo tanto, las hojas de cualquier árbol de testing tendrán predicados que son conjunciones de predicados atómicos.

Por otro lado, se mencionó que será imposible encontrar un caso de prueba abstracto para una clase de prueba cuando:

El predicado de la clase es una contradicción

Hay algún término indefinido en el predicado de la clase (tal es el caso de *KMR_SP_9* que aparece en la Figura 2.3).

Nuestro método trata a las contradicciones y a las indefiniciones de la misma manera, por lo cual en el resto del trabajo simplificaremos un poco la terminología y hablaremos solo de contradicciones para referirnos indistintamente a cualquiera de los 2 casos.

Partiendo de la base de que el predicado de una hoja en un árbol de testing es una conjunción de predicados atómicos, podemos decir que el predicado es una contradicción si y solo si:

Están presentes el predicado atómico p y el predicado atómico $\neg\, p$

Existe una contradicción matemática entre 2 o más predicados atómicos

El primer caso es sencillo de resolver, en tanto que el segundo es el centro de este trabajo.

Analicemos ahora un poco más las otras clases de prueba insatisfactibles mostradas en la Figura 2.3, para tener una idea un poco más profunda de cuáles son las contradicciones típicas que aparecen.

SCAddCat_SP_1 es insatisfactible porque la proposición $\{c?\} = \{\}$ es falsa. En *SCAddCat_SP_5* surge una contradicción en la conjunción de los predicados $categs \neq \{\}$ y $categs \subset \{c?\}$. En *Del_SP_1* surge una contradicción matemática ya que $p?$ no puede pertenecer a $\mathrm{dom}(procs \rhd \{waiting\})$ porque *procs* es un conjunto vacío. En *Deposit_SP_18* la aplicación $sa\ a?$ no está definida porque *sa* es vacía – notar que esta indefinición tiene una causa diferente a la indefinición en la clase *KMR_SP_9*, que también se puede ver en la Figura 2.3.

Como se puede observar, algunas de las contradicciones dependen de la semántica particular del *toolkit* matemático de Z, el cual es diferente de las semánticas de las teorías implementadas en algunos probadores de teoremas (por ejemplo en el asistente de pruebas Coq las funciones no

son conjuntos de pares ordenados). Esta situación implica que no es una tarea sencilla encontrar una herramienta existente, ya sea un SMT solver o un probador de teoremas, para resolver el problema de la poda utilizando la notación Z. Incluso, nuestro primer opción fue usar Z/EVES [16], pero probó ser menos efectivo y eficiente que el método que implementamos en Fastest, tal como se mostrará en el Capítulo 6.

Fastest provee un comando llamado `prunett` que analiza el predicado de cada hoja en el árbol de testing para determinar si el predicado es una contradicción o no. Teniendo en cuenta de que el problema es "no decidible", el algoritmo implementado en Fastest realiza el "mejor esfuerzo" y a la vez podrá mejorar su eficacia con la ayuda del usuario.

El núcleo del algoritmo es una librería que contiene los denominados *teoremas de eliminación*, cada uno de los cuales representa a una familia de contradicciones, representadas a través de uno o más predicados atómicos. Esta librería puede ser fácilmente extendida por los usuarios editando un archivo de texto. Por ejemplo, los siguientes teoremas de eliminación están incluidos en la librería.

ETheorem SingletonIsNotEmpty $[x : X]$

$\{x\} = \{\}$

ETheorem NotSubsetOfSingleton $[A : \mathbb{P}\, X;\ x : X]$

$A \neq \{\}$
$A \subset \{x\}$

Notar que la contradicción en *SCAddCat_SP_1* es una instancia de SingletonIsNotEmpty, mientras que NotSubsetOfSingleton generaliza la contradicción presente en *SCAddCat_SP_5*.

El pseudocódigo del algoritmo implementado para `prunett` se describe en la Figura 3.1 y se explicará más detalladamente en la Sección 3.1, pero primero es preciso introducir algunos conceptos del diseño.

LaTeX Los teoremas de eliminación en la librería están escritos con LaTeX utilizando el paquete CZT [13], el cual sigue el estándar ISO para la notación Z [10].

Parámetros Formales Cada teorema de eliminación posee un conjunto de parámetros formales, encerrados entre corchetes. Los parámetros pueden ser cualquier declaración de variables Z legal, y también se puede emplear la palabra reservada de Fastest `const` antes del nombre de un parámetro. Si un parámetro es precedido por `const` significa que Fastest podrá reemplazar dicho parámetro solo por constantes del tipo correspondiente. `const` se utiliza solo con parámetros de tipo $\mathbb{Z}$, $\mathbb{N}$ o cualquier tipo enumerado (tipos libres no inductivos). Cuando un teorema de eliminación contiene 2 o más parámetros constantes, estos pueden ser reemplazados solo por valores distintos. Por ejemplo, la librería contiene el siguiente teorema de eliminación:

ETheorem ExcludedMiddle $[x, \text{const } y, \text{const } z : X]$

$x = y$
$x = z$

Fastest podrá entonces aplicar este teorema de eliminación con $y \neq z$; por ejemplo, ExcludedMiddleEq (n,1,3) será una aplicación correcta pero no algo del estilo de ExcludedMiddleEq(n,3,3) o ExcludedMiddleEq(n,*count*,1).

Etapa de inicialización

1. Chequear los teoremas de eliminación y cargarlos
2. Aplicar reglas de equivalencia a los predicados atómicos de los teoremas
3. Obtener de cada teorema de eliminación las correspondientes expresiones regulares.

Cuando se ejecuta `prunett`

1. Podar clases de prueba con predicados de la forma $\cdots \wedge p \wedge \cdots \wedge \neg\, p \wedge \cdots$.
2. Para el predicado P de cada hoja en el árbol de testing:

   a) Convertir P a una cadena de caracteres.

   b) Para cada teorema de eliminación T en la librería, con parámetros formales $p_1 : T_1, \ldots, p_n : T_n$:

      1) Si el predicado de P hace pattern-matching con los predicados atómicos de T, luego:

         a' Si las expresiones de P que hicieron pattern-matching con los parámetros formales de T son del tipo o subtipo de $T_1, \ldots, T_n$ (es decir que pasan el chequeo de tipos), luego podar P y comenzar la siguiente iteración en 2.

      End for each.

   End for each.

Figura 3.1: prunett puede ser ejecutado luego de que los árboles de testing han sido creados.

El cuerpo de un teorema de eliminación El predicado de un teorema de eliminación debe ser una conjunción de predicados atómicos.

Un predicado atómico en un teorema de eliminación puede ser cualquier predicado atómico válido que utilice símbolos de Z soportados por Fastest, los nombres de los parámetros formales o de las palabras reservadas somewhere, anything y eval, las cuales serán explicadas luego.

Somewhere somewhere toma como parámetro una cadena de caracteres que representa una expresión Z. Por ejemplo, la librería contiene el siguiente teorema de eliminación:

ETheorem BasicUndefinition $[f : X \pfun Y;\ x : X]$

$x \notin \dom f$
somewhere$(f\ x)$

somewhere$(string)$ es similar a la expresión regular $*string*$[1]. Cuando el algoritmo encuentra esta directiva, intentará matchear con la expresión regular alguno de los predicados atómicos del predicado de la clase de prueba que está analizando.

Anything anything es equivalente a la expresión regular que hace match con cualquier cadena $(*)$. Por ejemplo, el siguiente teorema emplea esta directiva:

ETheorem SetNotASeq $[s : \seq X;\ n : \nat]$

$n = 0$
$s \neq \{\}$
$\dom s =$
$\quad \dom\{i : 1 \upto \text{anything} \bullet i + n - 1 \mapsto \text{anything}\}$

Dos o más ocurrencias de esta directiva pueden hacer match con diferentes cadenas.

Eval eval toma como argumento una expresión booleana constante y retorna *true* o *false* según corresponda. Llamamos expresión booleana constante a una expresión booleana que utiliza parámetros precedidos por const, valores de $\mathbb{Z}$, operadores Z o valores de tipos enumerados. El siguiente teorema de eliminación emplea esta directiva:

ETheorem RangeNotEmpty $[n, \text{const } N, \text{const } M : \nat]$

eval$(N \leq M)$
$n + N \upto (n + M) = \{\}$

Esta sentencia evalúa la expresión booleana; si es verdadera y los demás predicados atómicos del teorema hacen match con los de la clase que se está analizando, la clase será podada.

Reglas de Equivalencia Fastest aplica reglas de equivalencias, las cuales se toman de una librería que las contiene, a los teoremas de eliminación en las situaciones que correspondan. Esto implica, por ejemplo, que el ingeniero no necesitará escribir el siguiente teorema:

ETheorem NotInEmptyDom_Silly $[x : X;\ R : X \rel Y]$

[1] Para una completa descripción de la semántica ver [3]

Integers	$n < m$	$m > n$
	$n > m$	$m < n$
	$n \leq m$	$m \geq n$
	$n \geq m$	$m \leq n$
Sets	$A \cap B$	$B \cap A$
	$A \cup B$	$B \cup A$
All types	$x = y$	$y = x$
	$x \neq y$	$y \neq x$

(a) Equivalence rules.

Type	**Subtype**
$X \leftrightarrow Y$	$X \leftrightarrow Y$, $X \nrightarrow Y$, $X \rightarrow Y$, seq Y
$X \nrightarrow Y$	$X \nrightarrow Y$, $X \rightarrow Y$, seq Y
$\mathbb{Z}$	$\mathbb{Z}$, $\mathbb{N}$

(b) Subtyping rules

Figura 3.2: Fastest aplica por default estas reglas de equivalencia y reglas de subtipado.

$x \in \mathrm{dom}\, R$
$\mathrm{dom}\, R = \{\}$

porque la librería con los teoremas de eliminación ya contiene:

ETheorem NotInEmptyDom $[x : X;\ R : X \leftrightarrow Y]$

$x \in \mathrm{dom}\, R$
$R = \{\}$

y la librería con las reglas de reescritura posee $R = \{\} \Leftrightarrow \mathrm{dom}\, R = \{\}$. Las reglas de equivalencias son listas de predicados atómicos que deben ser equivalentes entre sí.

Además de las reglas de equivalencia presentes en la librería, Fastest aplica por default las reglas que figuran en la Tabla 3.2a.

Reglas de Subtipado Fastest también aplica algunas reglas simples de subtipado cuando se realiza la sustitución de los parámetros formales en un teorema de eliminación o en una regla de equivalencia. Una regla de subtipado determina cuando un tipo o un conjunto es subtipo de otro tipo o conjunto[2]. Por ejemplo, $X \rightarrow Y$ es subtipo de $X \nrightarrow Y$, el cual a su vez es subtipo de $X \leftrightarrow Y$.

Las reglas de subtipado empleadas en Fastest se listan en la Tabla 3.2b.

3.1. El Algoritmo

En esta sección explicaremos más detalladamente algunas partes de el pseudocódigo que se muestra en la Figura 3.1. El algoritmo fue implementado en Java y se basa fundamentalmente en el uso de expresiones regulares, pattern matching y búsqueda de cadenas. Solo en algunos pasos se hace uso de los objetos del AST de CZT, en particular, en el paso 1, en el chequeo de tipos y en el subtipado. Básicamente, cada teorema de eliminación es convertido de un conjunto de objetos Java a expresiones regulares de Java, y de manera análoga, el predicado de cada clase de prueba que se analiza es convertido a una cadena de caracteres[3]. Luego, se somete a un proceso de matching el predicado de la clase de prueba con las expresiones regulares asociadas

[2]Las nociones de tipo y subtipo en Z no son tan fuertes como en otros formalismos o herramientas, como Coq o PVS. Por esta razón no damos una definición más precisa de subtipado.

[3]En Java, expresiones regulares y cadena de caracteres son objetos diferentes

a un teorema de eliminación, tal como se explica detalladamente en la siguiente sección. Si se supera este paso, se realiza un chequeo de tipos de las expresiones que matchearon, los cuales serían los parámetros reales del teorema, para ver si concuerdan con los tipos de los parámetros formales del teorema. Si se supera esta última instancia, la clase será podada.

Durante la inicialización, Fastest carga la librería que contiene los teoremas de eliminación, los parsea, se construye un AST, aplica reglas de reescritura, se realiza el chequeo de tipos de los teoremas, empleando las herramientas provistas por el proyecto CZT [13].

En el paso 2 de la etapa de inicialización se aplican las reglas de equivalencia sobre los predicados atómicos de los teoremas de eliminación, obteniendo así expresiones regulares alternativas. Explicaremos más detenidamente este paso.

Como hemos dicho, las reglas de equivalencia son listas de predicados atómicos. Si $e_1, \ldots, e_n$ es una regla de equivalencia y hay un predicado atómico p en un teorema de eliminación que resulta ser una instancia de algún e_i, el algoritmo reemplazará p por $e_1 \mid \cdots \mid e_n$[4]—reemplazando oportunamente los parámetros formales en cada e_i por los parámetros formales de p. Por ejemplo, si la regla de equivalencia es $R = \{\} \Leftrightarrow \mathrm{dom}\, R = \{\}$ y el predicado atómico es $R \oplus G = \{\}$, el predicado se transformará en $R \oplus G = \{\} \mid \mathrm{dom}(R \oplus G) = \{\}$.

Una vez que la librería es exitosamente cargada, el usuario puede utilizar el comando `prunett`.

Como dijimos antes, si el predicado en forma de cadena de caracteres supera el proceso de matching con las expresiones regulares de un teorema de eliminación, comienza el proceso de *typechecking*. Este proceso consiste en tomar las cadenas de caracteres que se obtuvieron del proceso de matching que representan los parámetros reales del teorema, parsearlas con las herramientas provistas por CZT, y verificar que se adecúen al tipo de los parámetros formales del teorema. Este chequeo incluye las reglas de subtipado que ya hemos explicado. Entonces, si un teorema de eliminación tiene un parámetro de tipo U, cualquier expresión cuyo tipo es subtipo de U se considera que supera el chequeo.

3.2. Optimización en el proceso de pattern-matching

Las expresiones regulares se vuelven realmente complejas porque involucran *back references* (para asegurarnos de que cada parámetro formal se reemplace siempre por la misma expresión), expresiones regulares alternativas que provienen de las reglas de reescritura, etc. Por estos motivos, realizamos una serie de optimizaciones tendientes a mejorar la performance del algoritmo.

En primer lugar, y como paso previo al proceso de matching, se chequea si en la clase que se está analizando están presentes los operadores que forman parte de un teorema de eliminación en particular(el conjunto de operadores de un teorema viene dado por los operadores que figuran en su predicado, así como también los que se agregan tras aplicar reglas de reescritura), y en caso de que no estén presentes no se prosigue con el paso de matching pues se sabe de antemano que no habrá matching posible. Este paso ha repercutido en una mejora considerable en la eficiencia pues es mucho más rápido que el proceso de matching, ya que los operadores presentes en un teorema de eliminación están precalculados desde el momento de la carga y la obtención de los operadores en el predicado de una clase es un proceso trivial pues aprovecha la forma particular que se utiliza en el paquete de CZT para escribir operadores (todos comienzan con la barra invertida con excepción de cietos operadores matemáticos como +,-, etc.).

[4] $\mid$ es el operador "or" de expresiones regulares

Si se supera el paso anterior, se procederá a analizar si el predicado de la clase hace matching con las expresiones regulares asociadas al teorema de eliminación. Es en este punto en el que juegan un rol fundamental con respecto a la performance el uso que se haga de las expresiones regulares.

El paquete *java.util.regex* utiliza para el pattern-matching un tipo particular de Autómata Finito No-Determinístico (NFA) que puede convertirse en un gran cuello de botella si no se emplea adecuadamente. Es no determinístico porque cuando trata de hacer el matching de una expresión regular contra una cadena de caracteres, cada caracter en la cadena de entrada puede ser analizado en multiples ocasiones debido al algoritmo de *back tracking*.

Para nuestro problema en particular, crear una expresión regular única por cada teorema de eliminación tendrá un impacto tremendamente negativo en la performance porque hay ciertos factores que inciden en la eficiencia de el algoritmo de *back tracking*, en especial, el orden en el que aparecen los predicados atómicos y las formas alternativas de escribir un predicado atómico debido a la reescritura. Analicemos un caso muy sencillo, considerando un teorema de eliminación presente en la librería:

ETheorem ArithmIneq2 [const $N : \mathbb{N};\ n, m : \mathbb{Z}$]

$n \leq m$
$m < N$
$n > N$

Si intentamos crear una expresión regular para este teorema (luego de aplicar reescritura) sería algo del estilo:

```
((?:.*?$)*
^(.*)+ \leq (.*)+$ | ^(.*)+ \geq (.*)+$
(?:.*?$)*
^\2|\3 < (.*)$ | ^(.*) > \2|\3$
(?:.*?$)*
^\1|\4 > \5|\6$ | ^\5|\6 < \1|\4$
(?:.*?\$)*)
```

Y esto sin considerar las posibles permutaciones de orden de los 3 predicados atómicos involucrados, lo que nos dejaría una expresión regular 6 veces más grande.

Como se puede apreciar la expresión regular es verdaderamente compleja y va a tener una performance muy baja. En nuestro algoritmo utilizamos un enfoque diferente porque la eficiencia es un tema fundamental.

Lo que hacemos es crear una expresión regular por cada predicado atómico de un teorema de eliminación. Por ejemplo, para

```
n \leq m
```

sería

```
^(.*)+ \leq (.*)+$ | ^(.*)+ \geq (.*)+$
```

y empleamos una estructura auxiliar que mapea cada grupo de captura con un parámetro formal, para resolver el tema de las *back references*. Cuando se ejecuta `prunett`, obtenemos del

predicado de cada clase sus predicados atómicos e intentamos hacer matching con las expresiones regulares del teorema, procurando siempre reemplazar cada parámetro formal por la misma expresión valiendonos de la estructura auxiliar antes mencionada.

Este enfoque ha demostrado ser realmente eficiente y evita muchos de los problemas relacionados al *back tracking* porque maneja expresiones regulares mucho más pequeñas.

3.3. Poda Distribuída

Fastest es un sistema distribuido [2]. El usuario puede configurar la herramienta para distribuir algunas tareas. Cuando Fastest corre en una sola computadora, decimos que se está ejecutando en *modo aplicación* y cuando corre en más de una decimos que se está ejecutando en *modo distribuido*. Cuando ejecutamos en modo distribuido, `prunett` envía clases de prueba a diferentes servidores para realizar la poda en paralelo.

Al comienzo de nuestra investigación pensamos que la poda distribuida sería fundamental para obtener buenos valores de eficiencia. Sin embargo, el algoritmo demostró ser verdaderamente eficiente en modo aplicación (ver Capítulo 5) por lo cual debemos analizar la conveniencia de utilizar la poda distribuida, teniendo en cuenta factores tales como la penalización de tiempo debido a transmisiones a través de la red, sincronización, etc.

3.4. Discusión del método

Antes de implementar nuestro algoritmo de poda, aplicamos Fastest a algunos casos de estudio listados en la Tabla 5.1 en la página 20. Rápidamente observamos 3 hechos: (a) el TTF tiende a generar árboles de testing con un gran número de clases de prueba insatisfactibles; (b) todas ellas provienen de un conjunto no muy grande de contradicciones matemáticas; (c) los nuevos proyectos o las nuevas tácticas de testing pueden producir nuevos tipos de contradicciones. Inicialmente, consideramos la posibilidad de los SMT solvers pero fue imposible encontrar uno que implemente por completo el toolkit matemático de Z [15]. Entonces nuestro primer intento fue utilizar un probador de teoremas general como se sugiere en [8]. Utilizamos Z/EVES para podar los árboles de testing como se explicará en el Capítulo 5. Si bien funcionó razonablemente bien, no podó todas las clases insatisfactibles y demoró demasiado tiempo(ver la Tabla 6.1 en la página 23). Además, no resulta claro cómo los usuarios podrían extender el método sin conocimientos previos de probadores de teoremas.Por lo tanto, decidimos desarrollar un método específico para el TTF.

Desde el principio sabíamos que una solución completa sería imposible. Comenzamos a pensar entonces en una solución que *funcione en la práctica* aunque no sea sofisticada o elegante. Por "funcionar en la práctica" entendemos que la solución debería poder podar al menos el 80 % de las clases de prueba insatisfactibles que aparecen en especificaciones reales con mínima intervención del usuario. Es decir, una solución ingenieril, no necesariamente una solución formal y completa ya que, después de todo, estamos abordando el problema del testing. Además, si el método puede ser mejorado a medida que se consideran nuevos proyectos, el criterio de "funcionar en la práctica" debería ser alcanzado con el tiempo.

Nuestros resultados demuestran que el método presentado en este trabajo satisface, y tal vez excede, nuestro criterio de partida (ver Capítulo 5). De la descripción del método es fácil notar que a mayor cantidad de teoremas de eliminación, mayor cantidad de clases serán podadas por `prunett` sin intervención del usuario. De cualquier manera, uno se podría preguntar:

Cómo podemos saber si el método no funciona bien solo para los modelos con los que hemos experimentado? Es suficientemente general? Cuán generales son los teoremas presentes en la librería? Cuántas contradicciones pueden representar? Pensamos que la respuesta está en el hecho de que el lenguaje en el que escribimos los teoremas de eliminación es el mismo en el que están escritas las contradicciones que debemos detectar: la notación Z. Entonces, si los teoremas de eliminación y las contradicciones son expresadas en el mismo lenguaje, no hay razón para creer que alguna contradicción no podrá ser detectada. En el peor de los casos, los usuarios podrán agregar teoremas de eliminación que son exactamente iguales a las contradicciones que aparecen en las clases de prueba. Por ejemplo, si la contradicción en una clase de prueba es $x = 73 \land x = 12$, luego se podría agregar un teorema que sea:

ETheorem 73neq12 $[x : \mathbb{Z}]$

$x = 73$
$x = 12$

Más allá de esto, escribir tantos teoremas de eliminación como contradicciones aparezcan en cada proyecto, es tan impráctico como inspeccionar clases de prueba a mano. Como se observa en la Tabla 5.1 en la página 20, el método poda más de 2000 clases de prueba insatisfactibles con tan solo 52 teoremas. Esto fue posible porque el método le permite al usuario escribir teoremas de eliminación altamente parametrizados, transformandolos en patrones de contradicciones matemáticas.

Por último, vale la pena recalcar que el método no realiza ningún tipo de deducción como los probadores de teoremas. Esto quiere decir que si hay un teorema de eliminación T del cual la contradicción C es una instancia y a su vez la contradicción C' es deducible de C pero no es una instancia de T, se deberá agregar un nuevo teorema de eliminación para podar C'.

Capítulo 4

Certificación de los Teoremas de Eliminación

El usuario tiene la responsabilidad de mantener la librería de los teoremas de eliminación. Más allá de la flexibilidad que ofrece nuestro enfoque, esta posibilidad presenta un riesgo: el usuario puede añadir un teorema de eliminación que no representa una contradicción. Esta situación puede desembocar en la poda indebida de clases de prueba satisfactibles, lo que a su vez significará menos casos de prueba.

Para evitar este riesgo, los usuarios deberían probar formalmente que lo que ellos piensan que es una contradicción, realmente lo es. Este proceso es comúnmente denominado *certificación* porque involucra un asistente de pruebas.

En nuestro caso hemos utilizado Z/EVES para certificar 50 de los 52 teoremas de eliminación que figuran en la librería. Los teoremas que no fueron certificados hacen uso de la palabra reservada somewhere, cuya semántica no es sencilla de representar en Z. En el futuro investigaremos cómo certificar este tipo de teoremas también.

Debido a que la sintaxis de Z empleada por Z/EVES es diferente del estándar Z (el cual es empleado por Fastest), la presencia de las palabras reservadas eval y const, y el hecho de que un teorema de eliminación es precisamente lo opuesto a un teorema, tuvimos que hacer unos pequeños ajustes a la librería para poder exportarla a Z/EVES. Los cambios son los siguientes:

- Cambiar la forma en que los parámetros formales son escritos.
- Negar los predicados de los teoremas de eliminación.
- Agregar las condiciones incluídas en el operador eval como parte de los teoremas.
- Remover apropiadamente las ocurrencias del operador anything.
- Cuando se emplea const para 2 o más parámetros, agregar un predicado al teorema que indique que estas variables son distintas entre sí.

Por ejemplo, RangeNotEmpty (ver página 11) debe ser reescrito como sigue:

Theorem RangeNotEmpty
$$\forall\, n, N, M : \mathbb{N} \bullet \\ \neg\, (N \neq M \land N \leq M \\ \land\, n + N \mathinner{..} (n + M) = \{\})$$

Luego de exportar la librería cargamos los teoremas en Z/EVES y hacemos las 50 pruebas[1]. 33 (66 %) de las pruebas requirieron solo el comando `prove` (podemos decir que las pruebas fueron automáticas), tuvimos que agregar 5 lemas para probar 5 (10 %) de los teoremas porque eran bastante complejos, y los 12 restantes (24 %) requirieron más de un comando de prueba pero no fue necesario el uso de ningún lema.

Esta tipo de certificación de los teoremas de eliminación no es necesaria que sea empleada por los usuarios de Fastest. La incluimos en el trabajo por 2 razones fundamentales:

- Muestra otra área en la que los probadores de teoremas y el MBT pueden ser combinados.
- Muestra que la mayoría de los teoremas de eliminación son tan simples que pueden ser probados automáticamente.

Creemos que cualquier ingeniero de software que utilice notación Z puede fácilmente escribir otros teoremas de eliminación a medida que los necesite.

Una versión más avanzada de Fastestpodría incluir el proceso de certificación, ya sea bien como un servicio propio, o integrando con un probador de teoremas.

[1]Los teoremas junto con los scripts de prueba listos para ser cargados en Z/EVES están disponibles en `http://www.flowgate.net` en la sección de Herramientas.

Capítulo 5

Resultados empíricos

Básicamente creamos este método con un objetivo en mente: reducir el tiempo necesario para podar las clases de prueba insatisfactibles del árbol de testing producido por el TTF (en versiones anteriores de Fastest el proceso de poda lo realizaba manualmente el usuario con los comandos correspondientes). Para reducir este tiempo, la automatización y su performance fueron la clave.

Partiendo de que el problema es "no decidible", la automatización completa era imposible. Entonces, pensamos en un método que pueda crecer de proyecto en proyecto, a medida que el usuario va agregando nueva información. La simplicidad o la facilidad de uso, por lo tanto, se convirtieron en un factor fundamental.

Los experimentos que describiremos a continuación fueron realizados para medir si nuestro métodos satisfacía nuestras expectativas o no.En particular, corrimos 10 experimentos para medir:

- Cuanto tiempo de cómputo requiere `prunett` para podar los árboles de testing.
- Cuantos teoremas deben ser agregados de proyecto en proyecto en relación con las nuevas teorías matemáticas utilizadas en el nuevo proyecto.
- Cuan complejos pueden ser los teoremas de eliminación.

Estos parámetros fueron medidos al aplicar Fastest a los 10 casos de estudio que aparecen en la Tabla 5.1. 6 de ellos son ejemplos pequeños tomados de la literatura o propuestos por nosotros, mientras que los 4 restantes son ejemplos reales, provenientes de la industria. Estos casos de estudio provienen de 8 dominios de aplicación diferentes. Una descripción más detallada de los primeros 8 puede encontrarse en [2, página 179], el noveno es explicado en Sec. 6.2 y el último es similar a Plavis. En estos casos de estudio, los árboles de testing fueron construidos aplicando 2 o más tácticas de testing.

La Tabla 5.1 brinda tanto una idea de la complejidad de los modelos Z así como también los resultados de cada experimento. Cada columna representa lo siguiente[1]: **LOZC** proviene de líneas de código Z (en formato LaTeX); **State** indica el número de variables de estado; **Op** indica el número de operaciones Z involucradas en el experimento; **Clases** es el número de hojas del árbol de testing cuando se genera; **Atomic** es el número promedio de predicados atómicos presentes en las clases de prueba; **U** es el número de clases de prueba insatisfactibles (análisis manual); **Th** es el número *acumulado* de teoremas de eliminación necesarios para podar las

[1] Algunas se explican por sí mismas.

Caso de estudio	**R/T**	**LOZC**	**State**	**Op**	**Clases**	**Atomic**	**U**	**Th**	**Tiempo**	
Pool of sensors	Toy	46	1	1	15	100	8	7	0.5s	Desigualdad
Symbol table	Toy	78	1	3	26	100	16	16	0.6s	Teoría bás
Lift	Toy	152	6	3	17	100	1	17	0.5s	
Security class	Toy	172	4	7	36	100	16	17	0.8s	
Savings accounts	Toy	171	1	5	97	100	75	20	3s	Dominio d
Scheduler	Toy	240	3	10	213	100	164	33	7s	Cardinalid
Plavis	Real	608	13	13	232	100	50	38	15s	Secuencia
SWPDC	Real	1,238	18	17	201	100	56	52	31s	Rangos de
Steam boiler	Real	591	12	1	400	100	336	52	3s	
ECSS-E-70-41A	Real	774	13	5	1,226	100	856	52	2m18s	

Cuadro 5.1: Resumen de los experimentos. Todos los experimentos fueron realizados utilizando el mismo hardware y la misma plataforma de software: un procesador Intel Centrino Duo de 1.66 GHz con 1 Gb de memoria, corriendo sobre un Linux Ubuntu 8.04 con kernel 2.6.24-24-generic y empleando la máquina virtual Java SE Runtime Environment (build 1.6.0_14-b08). Fastest se utilizó en modo aplicación con el siguiente comando `java -Xss8M -Xms512m -Xmx512m -jar fastest.jar`.

clases de prueba insatisfactibles; **Tiempo** indica el tiempo necesario para podar; **Teorías** son las nuevas teorías matemáticas empleada para cada caso de estudio con respecto a los otros.

Para explicar un poco mejor el significado de la columna etiquetada como **Th**, consideremos lo siguiente.

Inicialmente, consideramos la librería con los teoremas de eliminación que contiene solo los siguientes 3 teoremas, los cuales están más relacionados con la lógica que con la matemática:

ETheorem NatDef $[n : \mathbb{N}]$

$\neg\ 0 \leq n$

ETheorem Reflexivity $[x, \text{const } y : X]$

$x \neq y$
$x = y$

ETheorem ExcludedMiddle $[x, \text{const } y, \text{const } z : X]$

$x = y$
$x = z$

Luego cargamos en Fastest el modelo más simple (Pool of sensors), aplicamos las tácticas correspondientes, generamos el árbol de testing, y finalmente ejecutamos `prunett` – llamamos a esto el *script*. Si `prunett` no pudo podar todas las clases de prueba insatisfactibles, extendemos la librería con la cantidad mínima de teoremas de eliminación para podar todas y luego corremos el script nuevamente. En otras palabras, con los 3 teoremas de eliminación que se encontraban inicialmente en la librería no se puedieron podar todas las clases de prueba insatisfactible por lo que hubo que agregar 4 teoremas nuevos, haciendo un total de 7. En este punto medimos

el tiempo de cómputo necesario para la poda. Este procedimiento fue repetido para todos los casos de estudio en orden creciente de complejidad (**ZLOC**), con la excepción de los últimos 2 casos que fueron ejecutados al final porque poseen muchas hojas y no agregan nuevas teorías matemáticas.

Las tablas 6.1 y 6.2 estudiadas en el Capítulo 6 dan más detalles de la performance de `prunett`.

Aunque estos experimentos deberían realizarse con un conjunto más grande de ejemplos, pensamos que muestra una buena tendencia que confirma nuestras expectativas. El tiempo de cómputo obtenido con este método es excelente comparado con el tiempo que demanda la inspección manual para encontrar casos insatisfactibles y con el tiempo que demandan enfoques similares — ver Capítulo 6. La segunda medida (que la intervención del usuario decrece a medida que nuevos proyectos son ejecutados) demuestra que nuestro método "funciona en la práctica".

Precisamente, la Tabla 5.1 confirma que cada vez que un modelo que no agrega nuevas teorías matemáticas es cargado en Fastest, casi ningún teorema de eliminación debe ser agregado.

Para determinar la facilidad de uso o la simplicidad del método analizamos la complejidad de los teoremas de eliminación. En la librería empleada para los experimentos el teorema de eliminación con predicado más grande posee solamente 3 predicados atómicos, aunque la mayoría posee 2. Esto no es más complejo que los teoremas presentes en las librerías de los probadores de teoremas. Afortunadamente, los predicados atómicos que aparecen en los teoremas de eliminación son simples generalizaciones de las contradicciones encontradas en las clases de prueba, por lo que el usuario no necesita hacer ningún tipo de inferencia lógica o deducción para poder escribirlos.

Capítulo 6

Comparación con Enfoques Similares

En este capítulo comparamos nuestro trabajo con un primer intento de utilizar el asistente de prueba Z/EVES, con los resultados reportados en [8], y con otros enfoques similares. Los puntos fundamentales a comparar son: la cantidad de clases inconsistentes que pueden ser podadas automáticamente, el tiempo mde cómputo requerido para la poda y la cantidad y simplicidad de los teoremas empleados.

6.1. Poda de Arboles de Testing con Z/EVES

Antes de implementar el método descripto en este trabajo utilizamos Z/EVES para podar los árboles de testing. Escribimos un script `bash` el cual toma el modelo Z y las clases generadas con Fastest y retorna una lista con las clases de prueba inconsistentes. En primer lugar, el script transforma los archivos al formato LaTeX empleado por Z/EVES[1]. Luego genera automáticamente un teorema de la forma:

Theorem UNSAT_TestClass

¬ TestClass

para cada clase de prueba y agrega el comando de prueba más poderoso, `prove by reduce`. Posteriormente, toda esta información es cargada en Z/EVES. Luego el script analiza la salida producida por Z/EVES, buscando aquellos teoremas que pudieron ser probados.

La Tabla 6.1 muestra los resultados de correr el script con los mismos casos de estudios analizados en el Capítulo 5. Debemos notar que el *toolkit* matemático empleado por Z/EVES contiene 565 teoremas y fastest apenas posee 52. Como se puede observar, Fastest tiene una mejor performance que Z/EVES en todos los puntos propuestos para comparar: poda más clases de prueba, en menos tiempo y con muchos menos teoremas.

6.2. Simplificación de Clases de Prueba con Isabelle

En [8] los autores aplican el probador de teoremas Isabelle para, entre otras cosas, eliminar las clases de prueba insatisfactibles. Ellos utilizan una codificación de Z en Isabelle, denominada HOL-Z [11], para testear la operación *STEAM_BOILER_WAITING* (*SBW*) del software de

[1]Las transformaciones solo funcionan para los casos de estudio, no fueron pensadas de manera general. Z/EVES no acepta el estándar ISO para Z.

Caso de Estudio	Z/EVES		Fastest	
	Podadas	**Time**	**Podadas**	**Time**
Pool of sensors	3	1s	8	0.5s
Symbol table	11	2s	16	0.7s
Lift	1	11s	1	0.5s
Security class	14	4s	16	0.8s
Savings accounts	45	15s	75	3s
Scheduler	123	26s	160	7s
Plavis	19	6m50s	50	16s
SWPDC	21	16m31s	56	31s
Steam boiler	159	23m9s	336	3s
ECSS-E-70-41A	728	1h20m51s	856	2m18s

Cuadro 6.1: Poda con Z/EVES.Los experimentos fueron realizados en la plataforma descripta en la Tabla 5.1.

Experimentos	HOL-Z			Fastest		
	DNF	**F**	**Tiempo**	**DNF**	**F**	**Tiempo**
SB	48	14	1m41s	50	14	0.7s
SBW	8	3	5s	8	8	0.5s
SBW DNFs unfolded	42	6	1m36s	impossible		
SBWdirect	384	?	22h	400	64	3s

Cuadro 6.2: Comparación con HOL-Z. Helke y sus colegas corrieron sus experimentos en una Sun Ultra-Sparc, mientras que nosotros lo hicimos en la plataforma descripta en la Tabla 5.1.

una caldera a vapor especificado en Z [1]. Los autores reconocen la necesidad de eliminar las clases de prueba luego de aplicar la táctica DNF y muestran los resultados en términos de el número de clases de prueba simplificadas y el tiempo de cómputo para realizar tal tarea en 4 experimentos con *SBW*. En la Tabla 6.2 reproducimos sus resultados y los nuestros luego de aplicar Fastest para los mismos experimentos.

El significado de las columnas es el siguiente: **DNF** representa el número de clases de prueba luego de aplicar la táctica DNF, **F** es el mismo número pero luego de podar (o simplificar) el DNF, y **Tiempo** es el tiempo empleado para la poda.

Por razones prácticas no incluimos el esquema Z de *SBW*, ya que éste incluye algunos esquemas (entre ellos el esquema de estado *SteamBoiler* (*SB*))que al ser expandidos derivan en un predicado muy grande. En el primer experimento los autores de [8] aplican DNF al esquema de estado *SB* sin expandir las referencias a esquemas que aparecen en él. En el segundo experimento aplican DNF a *SBW* sin expandir *SB*. En el tercero, expande el DNF de *SB* dentro del DNF de *SBW* produciendo las clases de prueba para este esquema. Esto es imposible de hacer en Fastest porque la simplificación es realizada luego de que el DNF del esquema más externo ha sido calculado. En el cuarto experimento se expande *SB* dentro de *SBW* e intenta computar las clases.

Como se puede apreciar en la Tabla 6.2, el tiempo de cómputo mostrado por Fastest es

sistemáticamente mucho mejor que los obtenidos por Helke y sus colegas. Aunque los experimentos realizados por Helke fueron realizados en una Sun Ultra-Sparc y los nuestros en una plataforma mucho más moderna, pensamos que la diferencia no proviene solamente de este hecho, particularmente en el cuarto experimento.

Por otra parte, nuestro método produce el mismo número de clases de prueba luego de la poda en el primer experimento, aunque el número de clases de prueba luego de calcular el DNF difiere. El número de clases de prueba simplificadas reportadas por Helke es extraño porque no lo hemos podido reproducir ni con Fastest, ni con Z/EVES e incluso ni a mano. Creemos que estas diferencias pueden provenir por usar diferentes versiones de la especificación[2]. En el cuarto experimento Helke no informa el número de clases de prueba luego de la simplificación. En este caso la diferencia en el número de clases de prueba luego de calcular el DNF proviene de la diferencia en el mismo número del primer experimento

In the fourth experiment Helke does no inform the number of test classes after simplification. In this case the difference in the number of test classes after DNF comes from the difference in the same number of the first experiment($384 = 48 \times 8$ and $400 = 50 \times 8$).

Considerando solo estos experimentos, podemos concluir que Fastest es más eficiente y al menos tan efectivo que el entorno HOL-Z en la simplificación de árboles de testing. Además, no es claro que HOL-Z pueda implementar fácilmente el TTF luego de calcular de las operaciones Z. Puede ser que la mejor performance de Fastest provenga de que que la herramienta fue concebida precisamente como una herramienta especializada TTF-MBT para la notación Z, mientras que HOL-Z es un entorno de pruebas Z, construido sobre un probador de teoremas general.

6.3. Otros Enfoques

No pudimos encontrar muchas otras referencias de trabajos que aborden la problemática de la poda en el contexto del testing basado en modelos. En sus trabajos, Stocks y Carrington advierten de que las ramas contradictorias deben ser removidas del árbol de testing [19]. Dick y Faivre [4] señalan que los sub-dominios contradictorios deben ser eliminados aplicando un conjunto de reglas extensibles.

Algunos otros trabajos reportan técnicas para simplificar tests de alguna manera, pero en diferentes contextos. Por ejemplo, en la posiblemente más completa y reciente investigación en el área de métodos formales y testing [9] se hace mención al tema de la poda de los árboles de testing. En [7] los autores utilizan heurísticas para reducir el espacio de búsqueda de secuencias de test en máquinas de estado finito abstractas. Meudec en su tesis de PhD [14] discute la simplificación para el lenguaje VDM. Doong y Frankl en [5][6] también simplifican las secuencias de test en el contexto de las especificaciones algebraicas de LOBAS.

[2]Nosotros trabajamos con una de los autores de [1].

Capítulo 7

Conclusiones y Trabajo Futuro

Hemos presentado un método alternativo a la prueba de teoremas para la poda de clases de prueba insatisfactibles de los árboles de testing, basado en la codificación de contradicciones en lugar de tautologías. Creemos que este método sigue las ideas de Dick y Faivre [4], quienes propusieron un conjunto extensible de reglas de eliminación.

La implementación presenta algunas características interesantes como performance, extensibilidad y simplicidad, confirmando nuestras creencias iniciales. De hecho, para nuestro poblema en particular, ha sido más útil y potente que un probador de teoremas en los casos de estudio que hemos analizado.

Como trabajo futuro, la principal tarea a desarrollar sería la integración con un probador de teoremas existente. Esto ayudaría a los usuarios a probar que los teoremas de eliminación son válidos, a probar propiedades de las especificaciones en el momento de la carga a Fastest, y mantener un conjunto mínimo de teoremas de eliminación en la librería.

Además, nos gustaría estudiar el algoritmo de la poda en modo distribuido, analizar en qué situaciones es mejor que modo aplicación, analizar su performance, etc.

Apéndice A

Documentación de Diseño

En este Apéndice incluiremos parte de la documentación del módulo encargado de la poda de árboles de testing en Fastest. Más precisamente, se brindará la documentación relativa a la Guía de Módulos así como también a la Estructura de Módulos, siguiendo los lineamientos planteados en .

Cabe destacar que si bien la documentación presentada aquí trata de reflejar fielmente la documentación original, se ha omitido la inclusión de algunos módulos más relacionados con la integración del módulo de poda con el resto de la herramienta que con la funcionalidad del mismo.

A.1. Estructura de Módulos

Module	**Pruning**
comprises	
Pruners	
Theorems	
TypeChecking	
Rewriting	
Operators	

Module	**Pruners**
comprises	
TreePruner	
TheoremsChecker	
PruneUtils	
ResultPrune	
PrePruner	
PruningVisitors	

Module	**Theorems**
comprises	
Theorem	
Variable	
TheoremsLoader	
TheoremsControl	

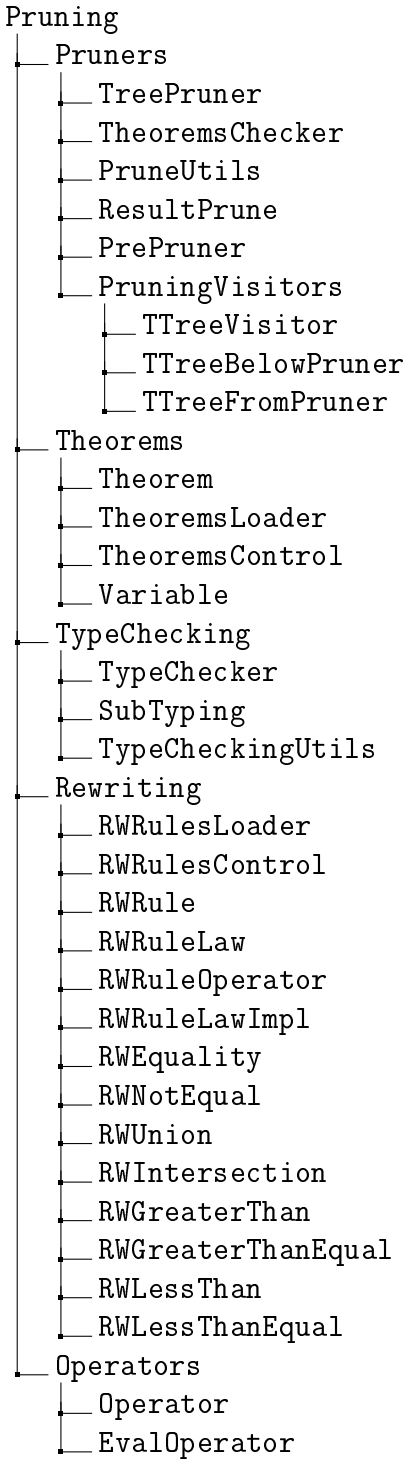
Module	**TypeChecking**
comprises	
TypeChecker	
SubTyping	
TypeCheckingUtils	

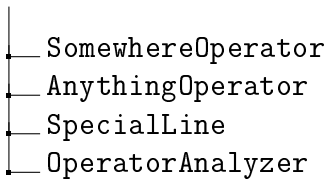
Module	**Rewriting**
comprises	
RWRulesLoader	
RWRulesControl	
RWRule	
RWRuleLaw	
RWRuleOperator	
RWRuleLawImpl	
RWEquality	
RWUnion	
RWIntersection	
RWNotEqual	
RWGreaterThan	
RWGreaterThanEqual	
RWLessThan	
RWLessThanEqual	

Module	**Operators**
comprises	
Operator	
EvalOperator	
SomewhereOperator	
AnythingOperator	
SpecialLine	
OperatorAnalyzer	

Module	**PruningVisitors**
comprises	
TTreeVisitor	
TTreeBelowPruner	
TTreeFromPruner	

A.2. Representación gráfica de la estructura de módulos





A.3. Guía de Módulos

1. **Pruning** Módulo lógico que contiene los módulos que implementan la poda del árbol de testing

   1.1. **Pruners** Módulo lógico que agrupa a los módulos que llevan adelante y coordinan la poda de las clases de prueba propiamente dicha.

      1.1.1. **TreePruner** Módulo físico que brinda funcionalidades para podar un subárbol o una clase en particular, asi como también posee un método para analizar si es posible eliminar una clase utilizando los teoremas de eliminación. Coordina todo el proceso de la poda automática con el método *pruneTClass()*. Interactúa con el módulo TheoremsChecker para que le encuentre teoremas de eliminación con los respectivos parámetros reales que hacen matching con el predicado de una clase de prueba que se está analizando y en caso de que la búsqueda sea satisfactoria interactúa con el módulo TypeChecker para que realiza el chequeo de tipos. Retorna una instancia de ResultPrune con la información y el resultado del análisis de poda. Oculta los algoritmos empleados para cumplir con su funcionalidad.

      1.1.2. **TheoremsChecker** Módulo físico que se encarga de obtener los teoremas cuya expresión regular hacen matching con el predicado de una clase que se le pasa como parámetro, a la vez que devuelve las cadenas de caracteres que se utilizaron para reemplazar a los parámetros formales del teorema. Interactúa con el módulo TheoremsControl para acceder a los distintos teoremas cargados en Fastest. Oculta los algoritmos y estructuras de datos empleados.

      1.1.3. **PruneUtils** Módulo físico que brinda funcionalidades comunes a los módulos que intervienen en el proceso de poda. Se encarga entre otras cosas de reemplazar parámetros formales por reales, de la búsqueda de cadenas dentro del predicado de las clases, etc. El módulo oculta los algoritmos que implementan tales tareas.

      1.1.4. **ResultPrune** Almacena la información relativa al análisis de poda de una clase en particular. Más precisamente, almacena el nombre de la clase, un valor booleano con el resultado del análisis y en caso de que la poda sea posible el nombre del teorema que determinó la poda y los parámetros reales empleados. Oculta las estructuras de datos empleadas.

      1.1.5. **PrePruner** Módulo físico que se encarga de analizar si en el predicado de una clase tiene la forma $\cdots \wedge p \wedge \cdots \wedge \neg\, p \wedge \cdots$. Oculta los algoritmos y estructuras de datos empleados.

      1.1.6. **PruningVisitors** Módulo lógico que agrupa aquellos módulos que implementen visitantes vinculados a la poda (patrón de diseño Visitor) sobre árboles de testing.

1.1.6.1. **TTreeVisitor** Interfaz que abstrae los algoritmos que implementan un visitante de árboles de prueba. Permite aplicar una operación particular a los nodos de un árbol de prueba, teniendo de que tipo particular es cada uno.

1.1.6.2. **TTreeBelowPruner** Módulo físico que permite recorrer un árbol de prueba para buscar una clase de prueba, especificada a través de su nombre en el constructor del módulo, para marcar sus clases hijas como podadas o como no podadas, según se especifique en el constructor del módulo. TTreeBelowPruner hereda de TTreeVisitor.

1.1.6.3. **TTreeFromPruner** Módulo físico que permite recorrer un árbol de prueba para buscar una clase de prueba, especificada a través de su nombre en el constructor del módulo, y marcarla como podada o no podada, según se especifique en el constructor del módulo. TTreeFromPruner hereda de TTreeVisitor.

1.2. **Theorems** Módulo lógico que agrupa a los módulos que almacenan información sobre los teoremas de eliminación, así como tambien engloba a los módulos afines.

1.2.1. **Theorem** Módulo físico que abstrae un teorema de eliminación. Almacena el nombre del teorema, los parámetros formales, el predicado, las expresiones regulares asociadas, etc. El módulo oculta las estructuras de datos empleadas para almacenar dicha información.

1.2.2. **TheoremsLoader** Módulo físico que se encarga de parsear el archivo de configuración que contiene la definición de los teoremas de eliminación, creando instancias de Theorem y cargándolas en la única instancia de TheoremsControl. Además tiene la responsabilidad de aplicar las reglas de reescritura sobre los predicados de los teoremas. El módulo oculta los algoritmos que implementan tales tareas.

1.2.3. **TheoremsControl** Módulo físico que almacena instancias de Theorem. Debe existir una única instancia por lo cual se emplea el patrón de diseño Singleton. El módulo oculta las estructuras de datos que emplea.

1.2.4. **Variable** Módulo físico que abstrae una variable de un teorema de eliminación. Almacena información relativa al nombre de la variable y su tipo. El módulo oculta las estructuras de datos empleadas para almacenar dicha información.

1.3. **TypeChecking** Módulo lógico que agrupa a aquellos módulos encargados de realizar el chequeo de tipos entre las expresiones contenidas en los predicados de las clases de prueba y los parámetros formales de los teoremas de eliminación.

1.3.1. **TypeChecker** Módulo físico que recibe como parámetros un teorema de eliminación, una clase de prueba y una lista de cadena de caracteres que se intentan utilizar como parámetros reales del teorema y se encarga de parsear dichas cadenas, analizar si se tratan de expresiones bien formadas dentro de la clase y posteriormente chequea si las expresiones son del tipo/subtipo adecuado, para lo cual interactúa con el módulo SubTyping. Retornará un valor booleano indicando si se ha pasado el chequeo de tipos.

1.3.2. **SubTyping** Módulo físico que se encarga de aplicar las reglas de subtipado, determinando cuando el tipo de una expresión es subtipo o no del tipo de otra expresión. Posee una variedad de métodos con diferentes firmas para adaptarse

a diferentes situaciones. Oculta los algoritmos empleados para cumplir con su funcionalidad.

1.3.3. **TypeCheckingUtils** Módulo físico que brinda funcionalidades relativas al chequeo de tipos a los demás módulos involucrados en dicho proceso. Oculta los algoritmos empleados para cumplir con su funcionalidad.

1.4. **Rewriting** Módulo lógico que agrupa a aquellos módulos encargados de reescribir los predicados de los teoremas de eliminación de una manera alternativa, utilizando reglas de reescritura, y módulos afines a dicha tarea.

1.4.1. **RWRulesLoader** Módulo físico que se encarga de parsear el archivo de configuración que contiene la definición de las reglas de reescritura, creando instancias de RWRule y cargándolas en la única instancia de RWRulesControl. El módulo oculta los algoritmos que implementan tales tareas.

1.4.2. **RWRulesControl** Módulo físico que almacena instancias de RWRule. Debe existir una única instancia por lo cual se emplea el patrón de diseño Singleton. El módulo oculta las estructuras de datos que emplea.

1.4.3. **RWRule** Interfaz de la cual heredan todas las reglas de reescritura. Los herederos deberán implementar el método *rewrite()* que toma como parámetro el predicado original y deberá devolver el predicado alternativo.

1.4.4. **RWRuleLaw** Interfaz de la cual heredan todas las reglas de reescritura relacionadas con leyes matemáticas que se encuentran en el archivo de configuración. Extiende la interfaz RWRule.

1.4.5. **RWRuleLawImpl** Módulo físico que implementa la interfaz RWRuleLaw. Los 2 métodos más importantes son *match()* y *rewrite()*. El método match() se emplea para determinar si un predicado satisface la ley y el método rewrite() devuelve la expresión alternativa. Oculta los algoritmos y estructuras de datos empleados.

1.4.6. **RWRuleOperator** Interfaz de la cual heredan todas las reglas de reescritura relacionadas con propiedades de operadores, tales como conmutatividad y asociatividad. Extiende la interfaz RWRule.

1.4.7. **RWEquality** Módulo físico que hereda de RWRuleOperator y se encarga de aplicar la propiedad de conmutatividad del operador de igualdad, devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.8. **RWUnion** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador de unión, devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.9. **RWIntersection** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador de intersección, devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.10. **RWNotEqual** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador de desigualdad, devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.11. **RWGreaterThan** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador "mayor que", devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.12. **RWLessThan** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador "menor que", devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.13. **RWGreaterThanEqual** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador "mayor o igual que", devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.4.14. **RWLessThanEqual** Módulo físico que se encarga de aplicar la propiedad de conmutatividad del operador "menor o igual que", devolviendo la expresión original y la que se obtiene de alterar el orden de los operandos.

1.5. **Operators** Módulo lógico que agrupa a los módulos que implementan los operadores propios de Fastest, así como también otros módulos afines.

1.5.1. **Operator** Interface de la cual heredarán todos los operadores de Fastest relacionados con la poda. Los herederos deberán implementar el método *addsemantic()*, el cual se encargará de transformar un predicado atómico en el que figura un operador en la expresión regular correspondiente, de acuerdo a la semántica de cada operador.

1.5.2. **AnythingOperator** Módulo físico que implementa el operador anything de Fastest. El operador se emplea para poder representar en un predicado en formato de texto una cadena que puede estar conformada por cualquier caracter. Se encarga de transformar un predicado atómico que contiene el operador anything en una expresión regular. Oculta los algoritmos empleados para satisfacer su funcionalidad.

1.5.3. **SomewhereOperator** Módulo físico que implementa el operador somewhere de Fastest. El operador se emplea para determinar si en un determinado ámbito se encuentra una cadena de caracteres. Se encarga de transformar un predicado atómico que contiene el operador somewhere en una expresión regular. Oculta los algoritmos empleados para satisfacer su funcionalidad.

1.5.4. **EvalOperator** Módulo físico que implementa el operador eval de Fastest. El operador se emplea para evaluar el valor de verdad de un predicado que solo contiene constantes numéricas. No crea ninguna expresión regular con el método *addsemantic()*, tan solo devuelve la misma cadena que recibió como parámetro. Oculta los algoritmos empleados para satisfacer su funcionalidad.

1.5.5. **SpecialLine** Módulo físico que almacena información acerca de un predicado atómico que contiene un operador propio de Fastest no relacionado con expresiones regulares, por lo que precisará un tratamiento especial. Hasta el momento solo se emplea con el operador eval. Los teoremas de eliminación almacenarán instancias de este objeto cada vez que aparezca un operador como los antes mencionados. Almacena el nombre del teorema al que pertenece, el predicado en cuestión y el operador involucrado. Oculta las estructuras de datos empleadas.

1.5.6. **OperatorAnalyzer** Módulo físico que se encarga de coordinar a los operadores de Fastest no relacionados con las expresiones regulares. Toma como parámetros una lista de instancias de objetos de tipo SpecialLine y una clase de prueba y de acuerdo al operador involucrado delega en el objeto que representa a dicho operador la responsabilidad de determinar si se satisface o no el predicado, retornando un valor booleano. Oculta los algoritmos empleados para satisfacer su funcionalidad.

Apéndice B

Teoremas de Eliminación

En este Apéndice se incluyen los teoremas de eliminación presentes en la actual versión de Fastest.

ETheorem NatDef $[n : \mathbb{N}]$

$\neg\ 0 \leq n$

ETheorem ExtensionalUndefinition $[f : X \nrightarrow Y;\ x : X]$

$x \notin \operatorname{dom} f$
$\mathsf{somewhere}(f\ x)$

ETheorem ArithmIneq1 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m < N$
$n = N$

ETheorem ArithmIneq2 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m < N$
$n > N$

ETheorem ArithmIneq3 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m = N$
$n > N$

ETheorem SingletonNotSet $[A : \mathbb{P}\, X;\ x : X]$

$x \notin A$
$\{x\} = A$

ETheorem BasicMembershipContradiction $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$x \notin A$

ETheorem NotInEmptySet $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$A = \{\}$

ETheorem SingletonIsNotEmpty $[x : X]$

$\{x\} = \{\}$

ETheorem NotSubsetOfSingleton $[A : \mathbb{P}\, X;\ x : X]$

$A \neq \{\}$
$A \subset \{x\}$

ETheorem NotSubsetOfSingletonMapplet $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$R \neq \{\}$
$\mathrm{dom}\, R \subset \mathrm{dom}\{x \mapsto y\}$

ETheorem SingletonNotSubset $[A : \mathbb{P}\, X;\ x : X]$

$x \notin A$
$\{x\} \subset A$

ETheorem DomNotSubsetOfSingleton $[R : X \leftrightarrow Y;\ x : X]$

$R \neq \{\}$
$\mathrm{dom}\, R \subset \{x\}$

ETheorem NotInEmptyDom $[R : X \leftrightarrow Y;\ x : X]$

$x \in \mathrm{dom}\, R$
$R = \{\}$

ETheorem BasicOrderingProperty $[n, m : \mathbb{Z}]$

$n \neq m$
$\neg\ n < m$
$\neg\ n > m$

ETheorem UndefinitionByEmptiness $[f : X \nrightarrow Y]$

$f = \{\}$
$\mathsf{somewhere}(f\ \mathsf{anything})$

ETheorem SingletonMappletNotInDom $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \operatorname{dom} R$
$\operatorname{dom}\{x \mapsto y\} = \operatorname{dom} R$

ETheorem SingletonNotSubsetDom $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \operatorname{dom} R$
$\operatorname{dom}\{x \mapsto y\} \subset \operatorname{dom} R$

ETheorem NrresEmptyRel $[R : X \leftrightarrow Y;\ A : \mathbb{P}\, Y;\ x : X]$

$x \in \operatorname{dom}(R \rhd A)$
$R = \{\}$

ETheorem NrresCap $[R : X \leftrightarrow Y;\ A : \mathbb{P}\, Y;\ x : X]$

$x \in \operatorname{dom}(R \rhd A)$
$\{x\} \cap \operatorname{dom} R = \{\}$

ETheorem CardDomEmptyRel $[R : X \leftrightarrow Y;\ \text{const}\ N, n : \mathbb{N}]$

$\mathsf{eval}(N > 0)$
$R = \{\}$
$n = N$
$\# \operatorname{dom} R = n$

ETheorem CardRelSingleton $[R : X \leftrightarrow Y;\ \text{const}\ N, n : \mathbb{N};\ r : X \times Y]$

$\mathsf{eval}(N > 1)$
$n = N$
$\# \operatorname{dom} R = n$
$\{r\} = R$

ETheorem SingletonMappletNotEqualRel1 $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \operatorname{dom} R$
$\{x \mapsto y\} = R$

ETheorem SingletonNotSubsetRel $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \operatorname{dom} R$
$\{x \mapsto y\} \subset R$

ETheorem NotInEmptyRan $[R : X \leftrightarrow Y;\ y : Y]$

$y \in \operatorname{ran} R$
$R = \{\}$

ETheorem SingletonMappletNotEqualRel2 $[R : X \leftrightarrow Y;\ x : X;\ \text{const}\ y1, \text{const}\ y2 : Y]$

$y1 \in \operatorname{ran} R$
$\{x \mapsto y2\} = R$

ETheorem BasicSetContradiction $[A : \mathbb{P}\, X]$

$A = \{\}$
$A \neq \{\}$

ETheorem ExcludedMiddleSingleton $[A : \mathbb{P}\, X;\ \text{const}\ x, \text{const}\ y : X]$

$\{x\} = A$
$\{y\} = A$

ETheorem ExcludedMiddleSingletonSub1 $[A : \mathbb{P}\, X;\ \text{const}\ x, \text{const}\ y : X]$

$\{x\} = A$
$\{y\} \subset A$

ETheorem ExcludedMiddleSingletonSub2 $[A : \mathbb{P}\, X;\ \text{const}\ x, \text{const}\ y : X]$

$\{x\} = A$
$A \subset \{y\}$

ETheorem RanNotSubsetOfSingleton $[R : X \leftrightarrow Y;\ y : Y]$

$R \neq \{\}$
$\operatorname{ran} R \subset \{y\}$

ETheorem SetComprNotEmpty1 $[\text{const}\ N : \mathbb{Z};\ A : \mathbb{P}\, X]$

$\mathsf{eval}(N < 2) A \neq \{\}$
$\{\mathsf{anything} : N \mathinner{..} \#A \bullet \mathsf{anything}\} = \{\}$

ETheorem SetComprNotASeq4 $[s : \operatorname{seq} X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\operatorname{dom} s = \operatorname{dom}\{i : 1 \mathinner{..} \mathsf{anything} \bullet i + n - 1 \mapsto \mathsf{anything}\}$

ETheorem SetComprNotASeq5 $[s : \operatorname{seq} X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\operatorname{dom}\{i : 1 \mathinner{..} \mathsf{anything} \bullet i + n - 1 \mapsto \mathsf{anything}\} \subset \operatorname{dom} s$

ETheorem NatRangeNotEmpty $[n, \text{const}\ N, \text{const}\ M : \mathbb{N}]$

$\mathsf{eval}(N \leq M)$
$n + N \mathinner{..} (n + M) = \{\}$

ETheorem ExcludedMiddle $[x, \text{const } y, \text{const } z : X]$

$x = y$
$x = z$

ETheorem SetComprIsEmpty1 $[R : X \leftrightarrow Y]$

$R = \{\}$
$\{\mathsf{anything} : \operatorname{dom} R \bullet \mathsf{anything}\} \neq \{\}$

ETheorem CapSubsetEmpty $[A, B : \mathbb{P}\, X]$

$A \neq \{\}$
$A \cap B = \{\}$
$A \subset B$

ETheorem CapEqEmpty $[A, B : \mathbb{P}\, X]$

$A \neq \{\}$
$A \cap B = \{\}$
$A = B$

ETheorem CapEmpty $[A, B : \mathbb{P}\, X]$

$A = \{\}$
$A \cap B \neq \{\}$

ETheorem NatRangeNotEmpty2 $[\text{const } N, \text{const } M : \mathbb{N}]$

$\mathsf{eval}(N < M)$
$N \mathinner{..} M = \{\}$

ETheorem NatRangeNotEmpty3 $[n, m, \text{const } N, \text{const } M : \mathbb{N}]$

$n + (m * N) \mathinner{..} (n + ((m + M) * N)) = \{\}$

ETheorem NatRangeNotSubset $[n, m, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$

$\mathsf{eval}(M - N > Q * P)$
$N \mathinner{..} M \subset n + (m * P) \mathinner{..} (n + ((m + Q) * P))$

ETheorem NatRangeNotEq $[n, m, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$

$\mathsf{eval}(M - N > Q * P)$
$N \mathinner{..} M = n + (m * P) \mathinner{..} (n + ((m + Q) * P))$

ETheorem NatRangeNotEmpty5 $[n, m, \text{const } P, \text{const } Q : \mathbb{N}]$

$\mathsf{eval}(Q * P > 0)$
$n + (m * P) \mathinner{..} (n + ((m + Q) * P)) = \{\}$

ETheorem NatRangeNotEmpty4 $[n, m : \mathbb{N}]$

$n \mathinner{..} (n + m) = \{\}$

ETheorem NatRangeNotSubset2 $[n, \text{const } N, \text{const } M, \text{const } P : \mathbb{N}]$

$\mathsf{eval}(M - N > P)$
$N \mathinner{..} M \subset (n \mathinner{..} (n + P))$

ETheorem NatRangeNotEq2 $[n, \text{const } N, \text{const } M, \text{const } P : \mathbb{N}]$

$\mathsf{eval}(M - N > P)$
$N \mathinner{..} M = (n \mathinner{..} (n + P))$

ETheorem NatRangeNotSubset3 $[n, m, p, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$

$\mathsf{eval}(M - N > Q * P)$
$m \leq (n + Q) * P$
$N \mathinner{..} M \subset p + (n * P) \mathinner{..} (p + m)$

ETheorem NatRangeNotEq3 $[n, m, p, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$

$\mathsf{eval}(M - N > Q * P)$
$m \leq (n + Q) * P$
$N \mathinner{..} M = p + (n * P) \mathinner{..} (p + m)$

Bibliografía

[1] Robert Büssow and Matthias Weber. A steam-boiler control specification with Statecharts and Z. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, pages 109–128, London, UK, 1996. Springer-Verlag.

[2] Maximiliano Cristiá and Pablo Rodríguez Monetti. Implementing and applying the Stocks-Carrington framework for model-based testing. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 167–185. Springer, 2009.

[3] Maximiliano Cristiá, Pablo Rodríguez Monetti, and Pablo Albertengo. The Fastest 1.5 User's Guide. Technical report, Flowgate Consulting, 2010.

[4] Jeremy Dick and Alain Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[5] Roong-Ko Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177, New York, NY, USA, 1991. ACM.

[6] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[7] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122, New York, NY, USA, 2002. ACM.

[8] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating Test Case Generation from Z Specifications with Isabelle. In *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1997.

[9] Robert M. Hierons and et al. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

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

[11] Kolyang, Thomas Santen, and Burkhart Wolff. A structure preserving encoding of Z in Isabelle/HOL. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 283–298, London, UK, 1996. Springer-Verlag.

[12] Ian Maccoll and David Carrington. Extending the Test Template Framework. In *Proceedings of the Third Northern Formal Methods Workshop*, 1998.

[13] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In *ZB. Lecture*, pages 65–84. Springer, 2005.

[14] C. Meudec. *Automatic generation of software tests from formal specifications.* PhD thesis, Queen's University of Belfast, Northern Ireland, UK, 1997.

[15] Mark Saaltink. The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada, 1997.

[16] Mark Saaltink. The Z/EVES System. In J.P. Bowen, M.G. Hinchey, and D. Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 72–85, 1997.

[17] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[18] P. Stocks. *Applying Formal Methods to Software Testing.* PhD thesis, Department of Computer Science, University of Queensland, 1993.

[19] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.