

Testing Basado en Especificaciones Z

Maximiliano Cristiá
Ingeniería de Software
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

2010

Resumen

En este apunte de clase se explica una técnica particular de testing basado en especificaciones Z. Esta técnica permite efectuar un testing muy completo, riguroso y disciplinado de un sistema del cual se posee un modelo Z.

No se incluyen ejercicios, para ello ver la práctica correspondiente.

Índice

1. Introducción	2
1.1. Programas y especificaciones	2
1.2. Casos de prueba exitosos, espacio válido de entrada y funciones de refinamiento y abstracción	2
1.3. El proceso de MBT	3
2. Generación de casos de prueba abstractos	3
3. Tácticas de MBT	6
3.1. El ejemplo	6
3.2. <i>IS</i> y <i>VIS</i> de <i>Extraer</i>	8
3.3. Forma Normal Disyuntiva (FND)	8
3.4. Particiones Estándar (PE)	10
3.5. Propagación de Sub-Dominios (PSD)	12
3.6. Mutación de Especificaciones (ME)	16
3.7. Causa-efecto (CE)	17
3.8. Otras tácticas	18
3.8.1. Tipos Libres (TL)	19
3.8.2. Límites de Implementación (LI)	19
4. El árbol de pruebas	20
5. Selección de casos de prueba abstractos	20

1. Introducción

En este capítulo veremos con detalle una técnica especial de testing basado en modelos (MBT). La técnica se denomina Test Template Framework (TTF) y fue propuesta por Phil Stocks y David Carrington a mediados de los noventa, e implementada de forma automática por primera vez recientemente en FastestTM por el autor, Pablo Rodríguez Monetti y Pablo Albertengo [CR09, CAR10]¹. El TTF se aplica especialmente a especificaciones Z, aunque puede utilizarse con otros formalismos.

Luego de abordar los aspectos teóricos de esta técnica los aplicaremos en la herramienta FastestTM la cual permite automatizar gran parte de la tarea.

1.1. Programas y especificaciones

Un programa es correcto si verifica su especificación. Por ende, para poder efectuar un testing significativo es necesario contar con alguna especificación del programa que se quiere probar. De lo contrario cualquier resultado que arroje el programa ante un caso de prueba no se sabrá si es correcto o no. La especificación puede ser de cualquier tipo (formal, semi-formal o informal) e incluso puede no estar escrita (la conoce el programador o el usuario). En algunos casos esta especificación está tan ausente o mal documentada que se la suele llamar *oráculo de decisión*, en clara alusión a algún mecanismo mágico que permite determinar si el programa es correcto o no. Este apunte, obviamente, presupone la existencia de una especificación o modelo formal Z del programa que se quiere testear. En Z la especificación de un programa es un esquema de operación.

1.2. Casos de prueba exitosos, espacio válido de entrada y funciones de refinamiento y abstracción

Diremos que un caso de prueba es exitoso si descubre un error en el programa. Pero para saber si es exitoso o no debemos recurrir a la especificación que es una descripción diferente del programa. Entonces surge el interrogante de saber cómo expresar los casos de prueba y los resultados del programa en términos de la especificación. De la misma forma que vemos a un programa como una función podemos ver a la especificación Z que le corresponde, es decir un esquema de operación, como otra función, **esta vez parcial**, que va desde el esquema definido por la declaración de las variables de entrada y de estado en el esquema definido por las variables de salida y de estado de la operación. Es decir, una especificación *Op* es una función **parcial** de *IS* en *OS*, donde *IS* se llama *espacio de entrada* y *OS* se llama *espacio de salida* y se definen de la siguiente forma²:

$$IS == [v?_1 : T_1, \dots, v?_a : T_a, s_1 : T_{a+1}, \dots, s_b : T_{a+b}]$$

$$OS == [v!_1 : U_1, \dots, v!_c : U_c, s_1 : T_{a+1}, \dots, s_b : T_{a+b}]$$

donde $v?_i$ son las variables de entrada, s_i las variables de estado y $v!_i$ las de salida utilizadas en *Op*. Notar que si *Op* es la especificación de *P* las dimensiones de *ID* e *IS* no tienen por qué ser iguales, lo que también vale para *OD* y *OS*. *Op* es una función parcial porque en general no todas las operaciones son totales; es decir, no todas las operaciones Z especifican qué ocurre para todas las combinaciones de los valores de las variables de entrada y estado. En consecuencia no tiene sentido testear un programa con un caso de prueba para el cual la especificación no es útil. Por lo tanto, definimos el *espacio válido de entrada* de la especificación *Op* como el subconjunto de *IS* que satisface la precondición de *Op*, formalmente:

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

¹FastestTM es marca registrada de Flowgate Consulting – Rosario, Argentina – <http://www.flowgate.net>.

²Notar que se usan términos diferentes para las entradas y salidas de los programas y las especificaciones.

lo que nos permite definir a Op como una función total [Sto93]:

$$Op : VIS_{Op} \rightarrow OS$$

Para poder formalizar la noción de caso de prueba exitoso nos hace falta aun una función que transforme elementos de VIS en elementos de ID y otra que haga lo propio entre OD y OS . Estas funciones se definen de la siguiente forma:

$$ref_P^{Op} : VIS_{Op} \rightarrow ID_P$$

$$abs_P^{Op} : OD_P \rightarrow OS_{Op}$$

Los nombres de las funciones refieren a que ref refina (es decir "des-abstrae") un elemento a nivel de la especificación en un elemento a nivel de la implementación; y, simétricamente, abs abstrae un elemento a nivel de la implementación en un elemento a nivel de la especificación. Por este motivo las llamaremos funciones de refinamiento y abstracción, respectivamente. abs se define como una función parcial pues no siempre será posible convertir la salida de un programa en un elemento de la especificación. Por ejemplo, si el programa termina en `Segmentation fault` generalmente no será posible abstraerlo a nivel de la especificación porque raramente una especificación considerará esta situación; lo mismo ocurre cuando el programa no termina o es tan erróneo que produce resultados totalmente inesperados. De hecho un programa que siempre emite salidas que pueden ser abstraídas es un programa más o menos correcto. Si bien en ninguno de estos casos se puede aplicar abs , esto no es un problema porque se ha logrado el objetivo: descubrir un error en P .

Con todos estos elementos podemos definir el concepto de caso de prueba exitoso. Sea P un programa tal que $P : ID \rightarrow OD$ y sea Op la especificación Z de P con $Op : VIS \rightarrow OS$; sean $t \in VIS$ y $x = ref_P^{Op}(t)$. (Nuevamente notar que tanto t como x son valores constantes, no variables ni parámetros.) Decimos que x es un caso de prueba exitoso para P sí y sólo sí $Op(t) \neq abs_P^{Op}(P(x))$. Conceptualmente, esta definición dice que lo que se esperaba que retornara P al suministrarle x no coincide con lo que su especificación indica. Una definición semejante puede darse si vemos a Op como un predicado lógico que depende de un elemento en VIS_{Op} y de uno en OS : x es exitoso para P sí y sólo sí $\neg Op(t, abs_P^{Op}(P(x)))$.

1.3. El proceso de MBT

El esquema mostrado en la Figura 1 grafica la situación anterior y muestra el proceso de testear un programa con técnicas de MBT. La especificación de la operación, Op se utiliza al comienzo y al final del ciclo: inicialmente se la utiliza para generar, gen , casos de prueba a nivel de la especificación (que llamaremos *casos de prueba abstractos*), t ; y finalmente se la utiliza como fórmula que determina si el caso de prueba es o no exitoso, $compr$. Luego de que se han generado los casos de prueba abstractos se utiliza la función ref para convertirlos en casos de prueba a nivel de la implementación (que llamaremos *casos de prueba concretos*), x . Estos casos son suministrados al programa P lo que produce una cierta salida $P(x)$ a nivel de la implementación. La función abs se utiliza para transformar $P(x)$ en un elemento a nivel de la especificación, y . Finalmente, t e y son reemplazados en Op para determinar si se ha descubierto un error o no.

Gran parte de los pasos del proceso de testing que mostramos se pueden automatizar considerablemente, como veremos al utilizar FastestTM. En este apunte se trata únicamente el paso denominado gen , es decir la generación de casos de prueba abstractos partiendo de una especificación Z .

2. Generación de casos de prueba abstractos

En esta sección veremos con más detalle el paso gen mostrado en la Figura 1. Para ello nos basaremos en los trabajos de Hörcher y Peleska [HP95], Stocks [Sto93] y Stocks y Carrington [SC96].

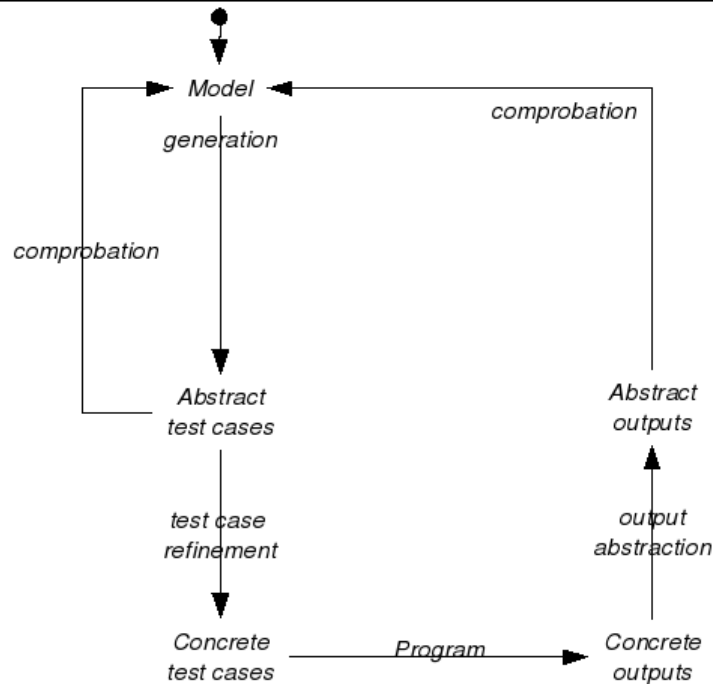


Figura 1: El proceso de testing basado en especificaciones formales.

La propuesta de estos autores es utilizar la especificación Z de una operación como fuente desde la cual obtener casos de prueba (abstractos) para testear el programa que supuestamente la implementa. Conceptualmente esta idea se basa en que la especificación del programa contiene todas las alternativas funcionales que el ingeniero consideró imprescindibles describir para que el programador implemente un programa correcto. Por lo tanto, para saber si el programa funciona correctamente es necesario probarlo para cada una de esas alternativas funcionales. Entonces, más concretamente, la técnica se basa en expresar cada una de esas alternativas funcionales (y otras más específicas de la implementación como veremos más adelante) como un esquema Z llamado *clase de prueba* incluido en el *VIS* de la operación; como las clases de prueba suelen formar una partición del *VIS* suelen llamarse clases de equivalencia. Estas clases se relacionan entre sí para formar lo que llamamos un *árbol de pruebas* [SC96].

La Figura 2 muestra esquemáticamente el proceso de generación de casos de prueba abstractos a partir de la especificación de una operación Op . Los primeros pasos consisten en definir el *IS* y el *VIS* de la operación. Luego se *intenta* dividir el *VIS* en clases de prueba aplicando una *táctica de testing*; esta división en ocasiones constituye una partición del *VIS* que a su vez es el primer nivel de nodos del árbol de pruebas³. Luego se aplica la misma u otras tácticas sobre una o varias de las clases de prueba del primer nivel del árbol de pruebas obteniendo el segundo nivel. Si bien se debe aplicar una única táctica sobre cada clase de prueba, no es obligatorio aplicar la misma táctica sobre todas las clases de prueba de un nivel dado. Al igual que con el *VIS*, la aplicación de una táctica de testing sobre una clase de prueba usualmente la divide en una serie de nuevas clases de prueba que constituyen una partición de aquella. Este proceso se repite hasta que el ingeniero de testing considere que en el último nivel del árbol están representadas todas las alternativas funcionales importantes de la operación; cada alternativa funcional corresponde a una única clase de prueba. El último paso del proceso es la selección de al menos un caso de prueba abstracto de cada clase de prueba del último nivel del árbol. En general no haremos más referencias en este apunte a este

³Stocks y Carrington usan el término *estrategia* en lugar de *táctica*. Nosotros creemos que *estrategia* refiere a algo de más largo alcance y global que el uso que hacen ellos. Por ejemplo, creemos que una *estrategia* de testing es el testing basado en modelos o el testing estructural. Por este motivo preferimos *táctica*.

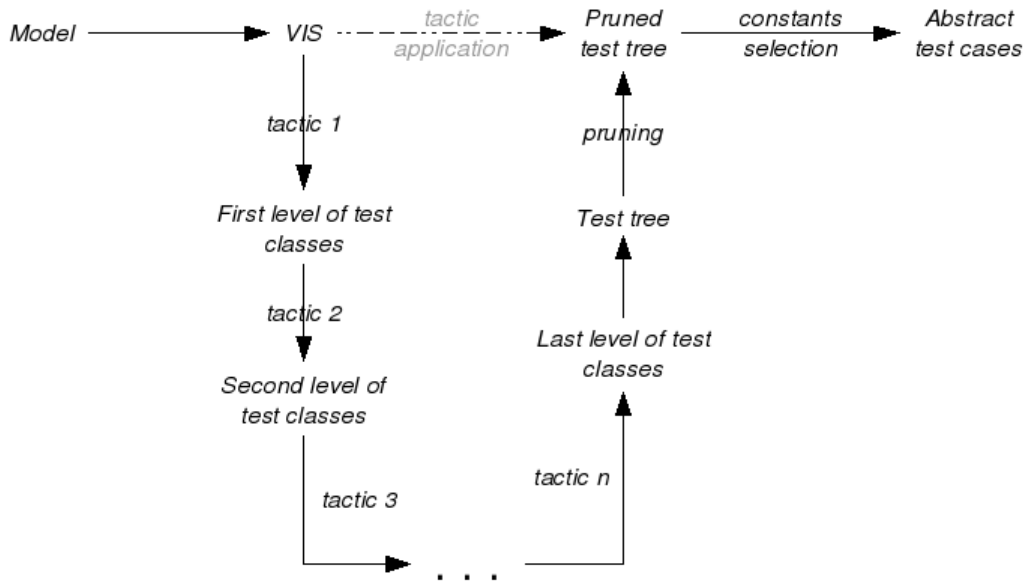


Figura 2: El proceso de generación de casos de prueba abstractos.

último paso pues, más allá de lo laborioso que puede ser, es muy simple ya que se trata de buscar valores para variables que reduzcan un predicado a verdadero. Vale la pena notar que este es uno de los pasos más difíciles de automatizar.

No siempre la aplicación de una táctica de testing genera una partición del *VIS* o de la clase de prueba sobre la cual se aplica. Sin embargo, la mayor parte de las tácticas están diseñadas para particionar el espacio en la mayor parte de los casos ya que esto brinda una mejor cobertura. En el peor de los casos se obtiene un *cubrimiento* del espacio.

El último nivel del árbol de pruebas es la familia de clases de prueba que no tienen hijos. Una clase de prueba no tiene hijos cuando no incluye ninguna otra clase de prueba (en términos de la representación gráfica usual, es un nodo hoja).

¿Cómo es que en cada clase de prueba se codifica una alternativa funcional? El *VIS* es el esquema *Z* definido por todas las variables de entrada y estado de la operación cuyos valores están restringidos por la precondition de la operación. Al aplicar una táctica de testing sobre el *VIS* se obtienen varios nuevos esquemas *Z* (las clases de prueba). Cada uno de estos esquemas se escribe de la siguiente forma:

$$Op_i^{Tac1} == [VIS \mid Q_i^{Tac1}]$$

donde *Tac1* es el nombre de la táctica aplicada y Q_i^{Tac1} es un predicado que de alguna manera representa la semántica de la táctica *Tac1* y es distinto para cada clase de prueba que la táctica genera. Notar que el predicado de *VIS* (que es la precondition de *Op*) queda conjugado con *Q* por lo que Op_i^{Tac1} es un subconjunto de *VIS*. En consecuencia Q_i^{Tac1} restringe los valores que pueden tomar las variables de entrada de la operación. Entonces, sea $x = ref_P^{Op}(t)$ con t en Q_i^{Tac1} . Como de alguna forma x verifica Q_i^{Tac1} , al ejecutar $P(x)$ el programa recorrerá un camino que debería verificar Q_i^{Tac1} . Ese camino representa una alternativa funcional del programa. Por ejemplo, si en la definición de *Op* hay una variable *A* de tipo $\mathbb{P}CHAR$, Q_i^{Tac1} es $A \neq \emptyset$ y *A* se refina a **struct** conjunto `{char A[MAX]; int ult;}`, entonces un caso de prueba abstracto posible es $A = \{ 'a', 'b' \}$ el cual refina a `{ 'a', 'b', '\0' }`, 2 y si *P* tiene una sentencia de la forma `if (A[0] == '\0') ...`; entonces *P* no entrará en la rama `then` del `if`: eso es una alternativa

funcional: se ejecutó la parte del programa que no procesa arreglos vacíos⁴.

La regla general es: *cuantas más clases de prueba haya y cuantos más niveles tenga el árbol de pruebas, más completo y confiable será el testing.*

Notar que este proceso puede aplicarse casi directamente sobre cualquier lenguaje de especificación basado en lógica como TLA o B, y puede adaptarse fácilmente a Statecharts [SMFM00] y CSP [PS97]. El resto del apunte trata sobre las diversas tácticas de testing y muestra cómo ir encadenándolas unas con otras para formar un árbol de pruebas.

3. Tácticas de MBT

Mostraremos las tácticas de testing reportadas en [Sto93, SC96] por medio de un ejemplo el cual finalizaremos mostrando el árbol de pruebas correspondiente y algunos casos de prueba abstractos. Tener en cuenta que lo que sigue puede aplicarse a otros lenguajes de especificación modificando levemente la notación. Además, observar que la mayoría de las tácticas presentadas pueden ser aplicadas automáticamente a cualquier especificación sin necesidad de intervención manual del ingeniero.

3.1. El ejemplo

Utilizaremos como ejemplo guía la especificación de la operación de extracción de dinero de una caja de ahorros. El banco permite que cada cliente tenga una única caja de ahorros y cada una de estas tiene un único titular. Los titulares se identifican con su DNI y el número de una caja de ahorro es el DNI de su titular. Sólo el titular puede extraer dinero de una caja de ahorros. Si bien no están permitidas extracciones que dejen la cuenta con un saldo inferior a cero, es posible que por otros motivos la cuenta tenga un saldo negativo (por ejemplo por un débito automático). No se permiten extracciones de más de \$1.000 a menos que el saldo sea superior a \$10.000. En consecuencia el modelo \mathbb{Z} es el que se muestra a continuación.

Comenzamos definiendo el tipo de los DNI de los clientes, las sumas de dinero como sinónimo de \mathbb{Z} y los mensajes de error que emitirá la operación *Extraer*.

[DNI]

DINERO == \mathbb{Z}

MENS ::= ok | error1 | error2

Las dos constantes mencionadas en los requerimientos las aislamos en sendas definiciones axiomáticas.

$extMax, saldoMin : \mathbb{N}$
$extMax = 1000$
$saldoMin = 10000$

Del banco únicamente consideramos sus cajas de ahorro las cuales las representamos con la función parcial ca de DNI en DINERO. Precisamente, si d es un DNI en el dominio de ca , $(ca\ d)$ es el saldo de la caja de ahorros d cuyo titular es la persona con DNI d . Podemos asumir que una caja de ahorros recién creada tiene saldo cero.

$CajasAhorroBanco$
$ca : DNI \rightarrow DINERO$

⁴Suponemos que el lenguaje de implementación es C.

Definimos tres posibles situaciones para la operación *Extraer*. La primera describe las condiciones en las que el titular puede extraer dinero. Deliberadamente escribimos los predicados usando un estilo que no es el habitual en estos cursos, pero la idea es poder mostrar ejemplos significativos de la aplicación de las tácticas de testing. Por ejemplo, el predicado de *ExtraerOk* contiene el operador \Rightarrow que no lo usamos regularmente. Por otro lado, se puede ver claramente que la forma de expresar los predicados no es ni siquiera rebuscada.

<i>ExtraerOk</i>
$\Delta CajasAhorroBanco$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$
$d? \in \text{dom } ca$ $m? > 0$ $m? \leq ca \ d?$ $m? > extMax \Rightarrow ca \ d? > saldoMin$ $ca' = ca \oplus \{d? \mapsto (ca \ d?) - m?\}$ $rep! = ok$

En un primer esquema de error capturamos las situaciones erróneas más triviales.

<i>ExtraerE1</i>
$\Xi CABanco$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$
$\neg (d? \in \text{dom } ca \wedge m? \leq ca \ d? \wedge m? > 0)$ $rep! = error1$

El segundo esquema de error muestra las condiciones en las cuales no le está permitida una extracción significativa a un titular.

<i>ExtraerE2</i>
$\Xi CABanco$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$
$d? \in \text{dom } ca$ $\neg (m? > extMax \Rightarrow ca \ d? > saldoMin)$ $rep! = error2$

Finalmente definimos la operación a testear, *Extraer*, como la disyunción de los tres esquemas anteriores⁵.

$$Extraer == ExtraerOk \vee ExtraerE1 \vee ExtraerE2$$

⁵En realidad no se testea la operación sino su implementación pero usamos la primera para guiar el testing de la última.

3.2. IS y VIS de Extraer

Como *Extraer* es una operación total el *IS* y el *VIS* coinciden:

$$VIS_{Extraer} == [ca : DNI \mapsto DINERO; d? : DNI; m? : DINERO]$$

es decir, es el esquema formado por la declaración de las variables de estado y de entrada usadas en la definición de la operación.

El *IS* y el *VIS* no coincidirían si, por ejemplo, la especificación no hubiera incluido los esquemas de error. En dicho caso tendríamos:

$$IS^\dagger == [ca : DNI \mapsto DINERO; d? : DNI; m? : DINERO]$$

$$VIS_{Extraer}^\dagger == [IS^\dagger \mid d? \in \text{dom } ca \wedge m? > 0 \wedge m? \leq ca \ d? \wedge m? > \text{extMax} \Rightarrow ca \ d? > \text{saldoMin}]$$

3.3. Forma Normal Disyuntiva (FND)

Esta suele ser la primera táctica que se aplica; no siempre genera una partición, lo que depende de la forma de la especificación. El primer paso es expresar la operación como una disyunción de esquemas en los cuales únicamente hay conjunciones de literales o de negaciones de literales, y el segundo paso es dividir el *VIS* con las precondiciones de cada esquema. El nombre de la táctica proviene de la lógica en donde expresar un predicado en FND significa transformarlo en un predicado formado por la disyunción de algún número de términos cada uno de los cuales contiene únicamente conjunciones de literales o negaciones de literales. Para llevar cualquier predicado a FND pueden aplicarse los algoritmos que se muestran en los libros de texto de lógica de primer orden, por ejemplo [Fit96], y también puede usarse un comando de prueba de Z/EVES que hace lo propio. En este apunte llevaremos *Extraer* a FND intuitivamente.

Como *Extraer* ya es una disyunción lo que debemos hacer es llevar cada uno de sus términos a FND. Entonces, en primer lugar debemos eliminar el \Rightarrow de *ExtraerOk*. Sabiendo que $a \Rightarrow b$ es lo mismo que $(\neg a \vee b)$ y que la conjunción es distributiva con respecto a la disyunción podemos transformar el esquema en dos esquemas en FND.

<i>ExtraerOk1</i>	<i>ExtraerOk2</i>
$\Delta CajasAhorroBanco$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$	$\Delta CajasAhorroBanco$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$
$d? \in \text{dom } ca$ $m? > 0$ $m? \leq ca \ d?$ $m? \leq \text{extMax}$ $ca' =$ $ca \oplus \{d? \mapsto (ca \ d?) - m?\}$ $rep! = ok$	$d? \in \text{dom } ca$ $m? > 0$ $m? \leq ca \ d?$ $ca \ d? > \text{saldoMin}$ $ca' =$ $ca \oplus \{d? \mapsto (ca \ d?) - m?\}$ $rep! = ok$

Continuamos con *ExtraerE1* en el cual hay que distribuir la negación en el paréntesis lo que implica generar los tres esquemas siguientes. Notar que en lo posible también eliminamos las negaciones reescribiendo los operadores relacionales y eliminamos las declaraciones de variables innecesarias (no es obligatorio pero mejora la legibilidad).

$\frac{\text{ExtraerE11}}{\exists CABanco}$ $d? : DNI$ $rep! : MENS$ <hr style="width: 100%;"/> $d? \notin \text{dom } ca$ $rep! = \text{error1}$	$\frac{\text{ExtraerE12}}{\exists CABanco}$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$ <hr style="width: 100%;"/> $m? > ca \ d?$ $rep! = \text{error1}$	$\frac{\text{ExtraerE13}}{\exists CABanco}$ $m? : DINERO$ $rep! : MENS$ <hr style="width: 100%;"/> $m? \leq 0$ $rep! = \text{error1}$
---	---	---

Finalmente toca el turno de eliminar el \Rightarrow del esquema *ExtraerE2* aplicando las mismas consideraciones que para *ExtraerOk*, pero esta vez solo debemos reescribir el mismo esquema debido a que la implicación está negada.

$\frac{\text{ExtraerE2}}{\exists CABanco}$ $d? : DNI$ $m? : DINERO$ $rep! : MENS$ <hr style="width: 100%;"/> $d? \in \text{dom } ca$ $m? > \text{extMax}$ $ca \ d? \leq \text{saldoMin}$ $rep! = \text{error2}$

De esta forma la operación queda expresada como se muestra a continuación:

$$\begin{aligned} \text{Extraer} == & \\ & \text{ExtraerOk1} \vee \text{ExtraerOk2} \\ & \vee \text{ExtraerE11} \vee \text{ExtraerE12} \vee \text{ExtraerE13} \\ & \vee \text{ExtraerE2} \end{aligned}$$

donde cada uno de los esquemas está en FND.

La aplicación de la táctica finaliza dividiendo el *VIS* con la precondition de cada esquema, lo que Stocks y Carrington recomiendan escribir de la siguiente forma.

$$\begin{aligned} \text{Extraer}_1^{FND} == & [VIS_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge m? > 0 \wedge m? \leq ca \ d? \wedge m? \leq \text{extMax}] \\ \text{Extraer}_2^{FND} == & [VIS_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge m? > 0 \wedge m? \leq ca \ d? \wedge ca \ d? > \text{saldoMin}] \\ \text{Extraer}_3^{FND} == & [VIS_{\text{Extraer}} \mid d? \notin \text{dom } ca] \\ \text{Extraer}_4^{FND} == & [VIS_{\text{Extraer}} \mid m? > ca \ d?] \\ \text{Extraer}_5^{FND} == & [VIS_{\text{Extraer}} \mid m? \leq 0] \\ \text{Extraer}_6^{FND} == & [VIS_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge m? > \text{extMax} \wedge ca \ d? \leq \text{saldoMin}] \end{aligned}$$

Primero, notar que si bien no se obtiene una partición del *VIS*, puesto que *Extraer*₁ y *Extraer*₂ se solapan, se logra un cubrimiento funcional básico. Segundo, notar que si tomamos un caso de prueba de cada una de las clases obtenidas estaremos probando el sistema en todas las situaciones más importantes:

1. Un titular quiere extraer una suma de dinero positiva pero no superior al saldo de su caja de ahorros ni al límite de extracción.

$S = \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \subset T$
$S = \emptyset, T \neq \emptyset$	$S \neq \emptyset, T \neq \emptyset, T \subset S$
$S \neq \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, T = S$
$S \neq \emptyset, T \neq \emptyset, S \cap T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \cap T \neq \emptyset, \neg(S \subseteq T), \neg(T \subseteq S), S \neq T$

Figura 3: Partición estándar para $S \cup T$, $S \cap T$ y $S \setminus T$.

2. Un titular quiere extraer una suma de dinero positiva pero no superior al saldo de su caja de ahorros en tanto esta tiene un saldo que supera el saldo mínimo para grandes extracciones.
3. Se intenta extraer con un *DNI* no registrado en el banco.
4. Un titular quiere extraer un monto superior al saldo de su cuenta.
5. El monto a extraer no es positivo.
6. Un titular quiere hacer una extracción de más de \$1.000 sin tener un saldo superior a \$10.000.

Por otro lado, se podría usar el operador pre de Z/EVES para escribir estas clases, por ejemplo:

$$Extraer_1^{FND} == [VIS_{Extraer} \mid \text{pre } ExtraerOk1]$$

aunque preferimos la otra notación porque deja explícita la condición. Es interesante observar que al escribir los invariantes de estado según el estilo TLA [Cri05] tenemos una ventaja en la aplicación de la táctica FND pues de lo contrario deberíamos haber calculado en primer término la precondition implícita de la operación.

Pero, ¿no habría que probar también qué ocurre cuando no hay cajas de ahorro abiertas, cuando hay una, cuando hay más de una, cuando el saldo es negativo, cero o positivo, etc.? Como es probable que estas pruebas también sean interesantes habría que combinarlas con las anteriores y habría que disponer de una táctica que permita generarlas. Precisamente, la combinación de pruebas se obtiene armando el árbol de pruebas y la siguiente táctica nos va a mostrar cómo hacer para generar los casos recién mencionados.

3.4. Particiones Estándar (PE)

En tanto que FND apunta a operar la especificación lógicamente, la táctica de Particiones Estándar trata con los operadores matemáticos de la operación. La idea es que cada operador matemático a nivel de especificación esconde una implementación compleja a nivel del programa. Entonces, dividir la especificación únicamente desde la lógica no es suficiente para revelar errores en la implementación de esos operadores matemáticos. Por lo tanto, se divide el dominio de cada operador simple (los complejos son tratados en la sección 3.5) asociándole una *partición estándar*. Una partición estándar es una partición del dominio del operador en conjuntos llamados *sub-dominos*; cada sub-dominio está definido por las condiciones que cada operando del operador debe cumplir. De esta forma cada sub-dominio se transforma en una condición para generar una clase de prueba. La Figura 3 muestra la partición estándar propuesta por Stocks en [Sto93] para los operadores \cup , \cap y \setminus . En [Sto93, Apéndice C] el autor propone particiones estándar para \subseteq , \subset y R^+ .

Aplicaremos esta táctica a las clases de prueba generadas en la sección 3.3. Como las particiones estándar presentadas en [Sto93] involucran operadores que no aparecen en nuestra especificación, en la Figura 4 proponemos una partición para los operadores relacionales definidos en \mathbb{Z} ($a <, \leq, =, \geq, > b$ con $a, b : \mathbb{Z}$), y en la Figura 5 una para los operadores \in y \notin . Obviamente algunos casos para

$a < 0, b < 0$	$a = 0, b < 0$	$a > 0, b < 0$
$a < 0, b = 0$	$a = 0, b = 0$	$a > 0, b = 0$
$a < 0, b > 0$	$a = 0, b > 0$	$a > 0, b > 0$

Figura 4: Partición estándar para los operadores relacionales definidos en \mathbb{Z} .

$A = \emptyset$	$A = \{b\}$	$A = \{b, c\}$	$\{b, c\} \subset A, a \notin A$
$A = \{a\}$	$A = \{a, b\}$	$\{a, b\} \subset A$	

Figura 5: Partición estándar para $a \in, \notin A$; se asume que a, b y c son tres elementos diferentes.

algunos operadores son contradictorios. Por otro lado, habría que probar formalmente que en efecto son particiones.

Para aplicar esta táctica primero hay que seleccionar *una aparición* en el esquema de operación de uno de los operadores. En el ejemplo guía inicialmente seleccionaremos el \leq o el $>$ que involucra a (*ca d?*) en $Extraer_i^{FND}$ con $i : 1, 2, 4, 6$; y el \notin de $Extraer_3^{FND}$; no aplicaremos la táctica en $Extraer_5^{FND}$ (simplemente para mostrar que no es necesario hacerlo en todas las clases). Lo único que hay que hacer en cada caso es reemplazar los parámetros formales de la descripción de las particiones por las expresiones usadas en la especificación y luego eliminar los casos contradictorios.

De esta forma obtenemos nuevas clases de prueba que se "cuelgan" de las anteriores al incluir aquellas en la definición de estas. Notar, por ejemplo, que aplicar la táctica a $Extraer_1^{FND}$ no tiene ningún efecto pues $m? > 0 \wedge m? \leq ca d?$ implican que $m? > 0 \wedge ca d? > 0$ lo que torna inútiles a todas las otras posibilidades de la partición estándar. Sólo la aplicación sobre $Extraer_i^{FND}$ con $i : 4, 6$ es significativa aunque no sobreviven todas las alternativas. Por lo tanto, ahora aplicaremos la partición estándar de \in a la expresión $d? \in \text{dom } ca$ de $Extraer_i^{FND}$ con $i : 1, 2$. En consecuencia se obtienen solo las siguientes clases.

$$\begin{aligned}
Extraer_1^{PE} &== [Extraer_1^{FND} \mid \text{dom } ca = \{d?\}] \\
Extraer_2^{PE} &== [Extraer_1^{FND} \mid \text{dom } ca = \{d?, d_1\} \wedge d_1 \neq d?] \\
Extraer_3^{PE} &== [Extraer_1^{FND} \mid \{d?, d_1\} \subset \text{dom } ca \wedge d_1 \neq d?]
\end{aligned}$$

$$\begin{aligned}
Extraer_4^{PE} &== [Extraer_2^{FND} \mid \text{dom } ca = \{d?\}] \\
Extraer_5^{PE} &== [Extraer_2^{FND} \mid \text{dom } ca = \{d?, d_1\} \wedge d_1 \neq d?] \\
Extraer_6^{PE} &== [Extraer_2^{FND} \mid \{d?, d_1\} \subset \text{dom } ca \wedge d_1 \neq d?]
\end{aligned}$$

$$\begin{aligned}
Extraer_7^{PE} &== [Extraer_3^{FND} \mid \text{dom } ca = \emptyset] \\
Extraer_8^{PE} &== [Extraer_3^{FND} \mid \text{dom } ca = \{d_1\} \wedge d_1 \neq d?] \\
Extraer_9^{PE} &== [Extraer_3^{FND} \mid \text{dom } ca = \{d_1, d_2\} \wedge \text{diff}(d_1, d_2, d?)] \\
Extraer_{10}^{PE} &== [Extraer_3^{FND} \mid \{d_1, d_2\} \subset \text{dom } ca \wedge \text{diff}(d_1, d_2, d?)]
\end{aligned}$$

$$\begin{aligned}
Extraer_{11}^{PE} &== [Extraer_4^{FND} \mid m? < 0 \wedge ca d? < 0] \\
Extraer_{12}^{PE} &== [Extraer_4^{FND} \mid m? = 0 \wedge ca d? < 0] \\
Extraer_{13}^{PE} &== [Extraer_4^{FND} \mid m? > 0 \wedge ca d? < 0] \\
Extraer_{14}^{PE} &== [Extraer_4^{FND} \mid m? > 0 \wedge ca d? = 0] \\
Extraer_{15}^{PE} &== [Extraer_4^{FND} \mid m? > 0 \wedge ca d? > 0]
\end{aligned}$$

$$\begin{aligned} \text{Extraer}_{16}^{PE} &== [\text{Extraer}_6^{FND} \mid ca \ d? < 0] \\ \text{Extraer}_{17}^{PE} &== [\text{Extraer}_6^{FND} \mid ca \ d? = 0] \\ \text{Extraer}_{18}^{PE} &== [\text{Extraer}_6^{FND} \mid ca \ d? > 0] \end{aligned}$$

Observar lo siguiente:

1. Términos como d_1 , d_2 y $diff$ deben definirse en definiciones axiomáticas o esquemas genéricos.
2. La inclusión de esquemas restringe la condición explícita de cada clase de prueba, por ejemplo la condición de Extraer_{16}^{PE} es en realidad $d? \in \text{dom } ca \wedge m? > \text{extMax} \wedge ca \ d? \leq \text{saldoMin} \wedge ca \ d? < 0$.
3. Cuando una de las condiciones exigidas por la partición estándar se deduce de las condiciones ya existentes entonces no es necesario escribirla, por ejemplo nunca incluimos $\text{saldoMin} > 0$ en Extraer_i^{PE} con $i : 16, 17, 18$ porque se deduce de su definición.
4. Si termináramos la generación de clases de prueba aquí tendríamos que ejecutar al menos 19 casos de prueba (Extraer_i^{PE} con $i : 1 \dots 18$ más Extraer_5^{FND}).
5. Si bien las nuevas clases no incorporan nuevas alternativas funcionales significativas (o al menos tan significativas como las de la sección 3.3), siguen especificando formas cada vez más particulares de ejecutar el programa. Por ejemplo, la alternativa especificada en Extraer_1^{FND} ahora se dividió en 3 casos:
 - a) Un titular quiere extraer una suma de dinero positiva pero no superior al saldo de su caja de ahorros ni al límite de extracción... cuando en el banco existe únicamente su cuenta.
 - b) Un titular quiere extraer una suma de dinero positiva pero no superior al saldo de su caja de ahorros ni al límite de extracción... cuando en el banco existen sólo dos cuentas, una de las cuales es la suya.
 - c) Un titular quiere extraer una suma de dinero positiva pero no superior al saldo de su caja de ahorros ni al límite de extracción... cuando en el banco existen más de dos cuentas, una de las cuales es la suya.

Claramente estos nuevos casos ejercitarán el programa en situaciones que antes podrían no haberse tenido en cuenta.

Suponiendo que tenemos una partición estándar para cada operador del lenguaje, ¿qué deberíamos hacer con los operadores que nosotros vamos definiendo a medida que extendemos el lenguaje? Stocks y Carrington proponen una aproximación diferente: sólo definimos particiones estándar para los operadores más simples del lenguaje y las particiones para los operadores que se definen en base a los simples se obtienen combinando esas particiones estándar. A esta técnica los autores la denominaron *propagación de sub-subdominios*.

3.5. Propagación de Sub-Dominios (PSD)

Como explicamos al finalizar la sección anterior, esta táctica en realidad no aporta ningún concepto nuevo pero es muy útil para obtener particiones estándar de operadores complejos combinando las particiones estándar de los operadores más simples. Se denomina Propagación de Sub-Dominios en virtud de que los sub-dominios de cada operador simple que participa en la definición del operador más complejo se propaga o combina con los sub-dominios de las particiones estándar de los otros operadores simples de la definición. El Cuadro 1 lista la definición de algunos operadores de Z en

Operador	Definición
$R \oplus G$	$(\text{dom } G \triangleleft R) \cup G$
$X \triangleleft R$	$R \setminus (X \triangleleft R)$
$R \triangleright Y$	$R \setminus (R \triangleright Y)$
$R(\mid X \mid)$	$\text{ran}(X \triangleleft R)$

Cuadro 1: La definición de algunos operadores de Z en términos de operadores simples.

función de otros más simples. Es posible que la definición dada en algunos libros sobre Z se dé en términos de conjuntos, cuantificaciones, etc.

La táctica también es útil para propagar particiones en expresiones en las que participan varios operadores (sean simples o complejos) pero que no necesariamente definen un nuevo operador, por ejemplo para:

$$H(\mid \text{dom}(X \triangleleft R) \mid)$$

se pueden propagar las particiones estándar de (\mid) y \triangleleft para obtener clases de prueba que contemplen diferentes alternativas para toda la expresión.

Supongamos que la definición del operador complejo, digamos \heartsuit , es⁶ : $\heartsuit(A, B, C) = (A \clubsuit B) \spadesuit C$, donde \clubsuit y \spadesuit son los operadores simples. Entonces la propagación de sub-dominios se efectúa de la siguiente manera:

1. Escribimos la partición estándar de cada operador que aparece en la expresión como la disyunción de sus reglas.

$$PE^\clubsuit(S, T) = D_1^\clubsuit(S, T) \vee \dots \vee D_n^\clubsuit(S, T)$$

$$PE^\spadesuit(U, V) = D_1^\spadesuit(U, V) \vee \dots \vee D_k^\spadesuit(U, V)$$

2. Comenzamos de adentro hacia afuera. Es decir, aplicamos PE^\clubsuit a la sub-expresión $(A \clubsuit B)$. Esto implica reemplazar los parámetros formales que aparecen en PE^\clubsuit por A y B respectivamente, y luego simplificar todo lo que sea posible. Puede ocurrir que durante la simplificación desaparezcan términos (por ejemplo si aplicamos la partición estándar de \cup a $A \cup A$ por lo menos desaparecen los términos 2 y 3). De esta forma obtenemos una disyunción de la forma:

$$PE^\clubsuit(A, B) = D_1^\clubsuit(A, B) \vee \dots \vee D_m^\clubsuit(A, B)$$

con $m \leq n$.

3. Luego hacemos lo mismo con PE^\spadesuit , obteniendo:

$$PE^\spadesuit(A \clubsuit B, C) = D_1^\spadesuit(A \clubsuit B, C) \vee \dots \vee D_j^\spadesuit(A \clubsuit B, C)$$

con $j \leq k$.

4. Finalmente, conjugamos las dos proposiciones así obtenidas y simplificamos (es decir, llevamos el resultado a FND, simplificamos cada término y eliminamos aquellos que reducen a false). Formalmente:

$$PE^\heartsuit = (\text{SIMPL} \circ \text{FND})(PE^\clubsuit(A, B) \wedge PE^\spadesuit(A \clubsuit B, C))$$

$R = \emptyset$	$R \neq \emptyset, X \neq \emptyset, X \cap \text{dom } R = \emptyset$
$R \neq \emptyset, X = \emptyset$	$R \neq \emptyset, X \cap \text{dom } R \neq \emptyset, \text{dom } R \subset X$
$R \neq \emptyset, X = \text{dom } R$	$R \neq \emptyset, X \cap \text{dom } R \neq \emptyset, \neg (\text{dom } R \subseteq X), \neg (X \subseteq \text{dom } R)$
$R \neq \emptyset, X \neq \emptyset, X \subset \text{dom } R$	

Figura 6: Nuestra partición estándar para el operador \triangleleft .

$R = \emptyset$	$R \neq \emptyset, \text{dom } G \neq \emptyset, \text{dom } G \subset \text{dom } R$
$R \neq \emptyset, \text{dom } G = \emptyset$	$R \neq \emptyset, \text{dom } G \neq \emptyset, \text{dom } G \cap \text{dom } R = \emptyset$
$R \neq \emptyset, \text{dom } G = \text{dom } R$	$R \neq \emptyset, \text{dom } G \cap \text{dom } R \neq \emptyset, \text{dom } R \subset \text{dom } G$
	$R \neq \emptyset, \text{dom } G \cap \text{dom } R \neq \emptyset, \neg (\text{dom } R \subseteq \text{dom } G)$
	$\neg (\text{dom } G \subseteq \text{dom } R)$

Figura 7: Resultado de la aplicación de la partición estándar de \triangleleft a $\text{dom } G \triangleleft R$; notar que no se elimina ningún caso.

Aplicaremos la táctica al operador \oplus que aparece en *ExtraerOk1* y *ExtraerOk2*. El primer paso para poder hacerlo es obtener una partición estándar del operador \triangleleft dado que aparece en la definición de \oplus . En [SC96] los autores sugieren una partición estándar para este operador pero nosotros la modificamos levemente dividiendo el último caso en dos como se muestra en la Figura 6. Entonces usando esta partición y la de \cup mostrada en la Figura 3 debemos aplicar el procedimiento antedicho para obtener la partición estándar para el operador \oplus . El primer paso podemos asumir que lo tenemos. El segundo paso, es decir aplicar la partición estándar de \triangleleft a $(\text{dom } G \triangleleft R)$, se muestra en la Figura 7.

El tercer paso se muestra en la Figura 8; esto es, reemplazar $(\text{dom } G \triangleleft R)$ por S y G por T en la Figura 3. Notar que aquí se pierden los últimos cuatro casos de la partición estándar de \cup (ver Figura 3) debido a que $(\text{dom } G \triangleleft R) \cap G = \emptyset$ en tanto que en esos casos se pide (de una u otra manera) que la intersección sea no vacía.

Finalmente, calculamos el "producto" de las Figuras 7 y 8 seguido de la simplificación y el descarte de los casos contradictorios. El resultado es la partición estándar para \oplus mostrada en la Figura 9; observar la similitud con la partición estándar de \cup (Figura 3). Analicemos algunos casos descartados. Los casos 1-c y 1-d se descartan porque si $R = \emptyset$, $(\text{dom } G \triangleleft R)$ no puede ser distinto de vacío. El caso 2-a se descarta porque si $\text{dom } G = \emptyset$, entonces $\text{dom } G \triangleleft R = R$ pero $R \neq \emptyset$ por lo tanto $(\text{dom } G \triangleleft R)$ no puede ser vacío. En el caso 3-a la contradicción se da porque si $R \neq \emptyset$ y $\text{dom } G = \text{dom } R$ entonces $G \neq \emptyset$ pero a pide $G = \emptyset$.

Ahora que tenemos la partición estándar de \oplus aplicamos el mismo procedimiento que aplicamos con cualquier partición estándar, es decir, reemplazamos R por ca y G por $\{d? \mapsto (ca \ d?) - m?\}$ en la Figura 9 y simplificamos obteniendo los casos de la Figura 10. Obviamente desaparecen varios casos porque G es reemplazado por un conjunto unitario (trivialmente no vacío), lo que a su vez nos permite simplificar las condiciones (por ejemplo, donde dice $G \neq \emptyset$ no ponemos nada). Los cuatro casos que sobreviven indican que podremos dividir cada una de las clase de prueba que tenemos hasta el momento en cuatro (aunque en algunos casos algunas de las nuevas clases generadas desapareceran porque las nuevas condiciones entran en contradicción con las que ya estaban o debido a que las nuevas condiciones ya estaban). Observar el significado conceptual de cada caso en relación a los distintos

⁶Para simplificar la presentación supongamos que sólo participan dos operadores **binarios** simples. Si en lugar de la definición de un operador complejo tenemos simplemente una expresión que involucra dos operadores simples, el procedimiento es el mismo.

$$\begin{aligned}
(\text{dom } G \triangleleft R) &= \emptyset, G = \emptyset \\
(\text{dom } G \triangleleft R) &= \emptyset, G \neq \emptyset \\
(\text{dom } G \triangleleft R) &\neq \emptyset, G = \emptyset \\
(\text{dom } G \triangleleft R) &\neq \emptyset, G \neq \emptyset, (\text{dom } G \triangleleft R) \cap G = \emptyset
\end{aligned}$$

Figura 8: Resultado de aplicar la partici3n estandar de \cup a $\text{dom } G \triangleleft R$ y G .

$$\begin{array}{ll}
R = \emptyset, G = \emptyset & R \neq \emptyset, G \neq \emptyset, \text{dom } G \subset \text{dom } R \\
R = \emptyset, G \neq \emptyset & R \neq \emptyset, G \neq \emptyset, \text{dom } R \cap \text{dom } G = \emptyset \\
R \neq \emptyset, G = \emptyset & R \neq \emptyset, G \neq \emptyset, \text{dom } R \subset \text{dom } G \\
R \neq \emptyset, G \neq \emptyset, \text{dom } R = \text{dom } G & R \neq \emptyset, G \neq \emptyset, \text{dom } R \cap \text{dom } G \neq \emptyset, \\
& \neg (\text{dom } G \subseteq \text{dom } R), \neg (\text{dom } R \subseteq \text{dom } G)
\end{array}$$

Figura 9: Partici3n estandar para el operador \oplus obtenida por propagaci3n entre \cup y \triangleleft .

estados en los que pueden estar las cajas de ahorro del banco:

1. El banco no tiene ninguna caja de ahorros.
2. El banco tiene una 3nica caja de ahorros que es aquella de la que el cliente quiere extraer el dinero.
3. El banco tiene al menos dos cajas de ahorros una de las cuales es aquella de la que el cliente quiere extraer el dinero.
4. El banco tiene al menos una caja de ahorros pero no es aquella de la que el cliente quiere extraer el dinero.

La nuevas clases de prueba, que se listan a continuaci3n, m1s las que ya ten1amos suman 33 (lo que implica que deberemos testear *Extraer* con al menos 33 casos de prueba).

$$\begin{aligned}
\text{Extraer}_1^{\text{PSD}} &== [\text{Extraer}_{11}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_2^{\text{PSD}} &== [\text{Extraer}_{11}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\text{Extraer}_3^{\text{PSD}} &== [\text{Extraer}_{12}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_4^{\text{PSD}} &== [\text{Extraer}_{12}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\\
\text{Extraer}_5^{\text{PSD}} &== [\text{Extraer}_{13}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_6^{\text{PSD}} &== [\text{Extraer}_{13}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\text{Extraer}_7^{\text{PSD}} &== [\text{Extraer}_{14}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_8^{\text{PSD}} &== [\text{Extraer}_{14}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\\
\text{Extraer}_9^{\text{PSD}} &== [\text{Extraer}_{15}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_{10}^{\text{PSD}} &== [\text{Extraer}_{15}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\text{Extraer}_{11}^{\text{PSD}} &== [\text{Extraer}_{16}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_{12}^{\text{PSD}} &== [\text{Extraer}_{16}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\\
\text{Extraer}_{13}^{\text{PSD}} &== [\text{Extraer}_{17}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_{14}^{\text{PSD}} &== [\text{Extraer}_{17}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca] \\
\text{Extraer}_{15}^{\text{PSD}} &== [\text{Extraer}_{18}^{\text{PE}} \mid \text{dom } ca = \{d?\}] \\
\text{Extraer}_{16}^{\text{PSD}} &== [\text{Extraer}_{18}^{\text{PE}} \mid \{d?\} \subset \text{dom } ca]
\end{aligned}$$

$$\begin{aligned}
\text{dom } ca &= \{d?\} \\
\{d?\} &\subset \text{dom } ca \\
ca &\neq \emptyset, \text{dom } ca \cap \{d?\} = \emptyset
\end{aligned}$$

Figura 10: Resultado de aplicar la partición estándar de \oplus a la expresión $ca \oplus \{d? \mapsto (ca \ d?) - m?\}$.

3.6. Mutación de Especificaciones (ME)

Esta táctica bien utilizada es muy poderosa, el problema es que no es simple usarla correctamente y, menos aun, automatizarla en gran medida. La idea fundamental detrás de la Mutación de Especificaciones es que el ingeniero a cargo del testing escriba la especificación de la operación que él intuye que el programador programó (ya que probablemente haya cometido un error y no haya programado lo que dice la especificación original), y que luego genere un caso de prueba que *distinga* las dos especificaciones. Esta nueva especificación que escribe el ingeniero testeador se dice que es un *mutante* de la especificación original. En otras palabras un mutante es una especificación que tiene alguna diferencia con la original, diferencia que intenta capturar el posible error cometido por el programador. Cada mutante puede aplicarse sobre una clase de prueba diferente del último nivel del árbol de pruebas o el mismo puede aplicarse sobre todas las clases y otro mutante puede aplicarse sobre todo o una parte del nivel siguiente.

Por ejemplo las dos especificaciones que siguen son mutantes de *ExtraerOk*.

<u><i>M1_ExtraerOk</i></u>	<u><i>M2_ExtraerOk</i></u>
$\Delta \text{CajasAhorroBanco}$ $d? : \text{DNI}$ $m? : \text{DINERO}$ $\text{rep!} : \text{MENS}$	$\Delta \text{CajasAhorroBanco}$ $d? : \text{DNI}$ $m? : \text{DINERO}$ $\text{rep!} : \text{MENS}$
$d? \in \text{dom } ca$ $m? > 0$ $m? \leq ca \ d?$ $m? > \text{extMax} \Rightarrow ca \ d? > \text{saldoMin}$ $ca' = ca \cup \{d? \mapsto (ca \ d?) - m?\}$ $\text{rep!} = \text{ok}$	$d? \in \text{dom } ca$ $m? > 0$ $m? \leq ca \ d?$ $m? > \text{extMax} \vee ca \ d? > \text{saldoMin}$ $ca' = ca \oplus \{d? \mapsto (ca \ d?) - m?\}$ $\text{rep!} = \text{ok}$

Es decir que en el primer caso el ingeniero testeador intuye que el programador confunde \oplus con \cup , en tanto que en el segundo intenta ver si el programador confunde \Rightarrow con \vee .

Continuando con la generación de casos de prueba para *Extraer*, aplicaremos la táctica usando *M2_ExtraerOk* (el lector puede hacer lo propio con *M1_ExtraerOk*). Según explicamos más arriba, ahora debemos generar un caso de prueba que distinga *ExtraerOk* de *M2_ExtraerOk*. Dado que el mutante cambia un operador lógico por otro, buscamos la diferencia analizando las tablas de verdad respectivas, ver Tabla 2.

Como *M2_ExtraerOk* se diferencia de *ExtraerOk* si $m? > \text{extMax}$ y $ca \ d? > \text{saldoMin}$ asumen los valores de verdad de la segunda y última filas de la Tabla 2 para p y q , respectivamente, entonces definimos el siguiente conjunto:

$$\begin{aligned}
\text{DEC_M2_ExtraerOk} &== \\
&[\text{VIS}_{\text{Extraer}} \mid \\
&\quad m? > \text{extMax} \wedge ca \ d? \leq \text{saldoMin} \\
&\quad \vee m? \leq \text{extMax} \wedge ca \ d? \leq \text{saldoMin}]
\end{aligned}$$

$m? > extMax$	$ca d? > saldoMin$	\Rightarrow	\vee
t	t	t	t
t	f	f	t
f	t	t	t
f	f	t	f

Cuadro 2: Los operadores se diferencian en la segunda y última filas.

que distingue ambas especificaciones en el sentido de que⁷

$$DEC_M2_ExtraerOk == VIS_{Extraer} \setminus pre (ExtraerOk \cap M2_ExtraerOk)$$

Por lo que tomando cualquier representante de $DEC_M2_ExtraerOk$ podremos ver si P implementa $M2_ExtraerOk$ o no. El nombre DEC proviene de *distinguishing equivalent class*. Notar, sin embargo, que hay varias situaciones en las cuales la táctica no aporta nada significativo. Si $ExtraerOk \cap M2_ExtraerOk = \emptyset$ entonces $DEC_M2_ExtraerOk$ coincidirá con el VIS lo que no aportará ninguna condición nueva para el testing; algo similar ocurre si $ExtraerOk \subset M2_ExtraerOk$. Por el contrario si $M2_ExtraerOk \subset ExtraerOk$ entonces $DEC_M2_ExtraerOk$ estará restringido por un predicado que puede aportar algo al testing.

El último paso en la aplicación de la táctica consiste en particionar una o más clases de prueba del último nivel del árbol usando el predicado de $DEC_M2_ExtraerOk$ y su negación ($\neg DEC_M2_ExtraerOk$). En este ejemplo aplicaremos la táctica sobre las clases $Extraer_3^{PE}$ y $Extraer_6^{PE}$; notar que no son clases obtenidas en el último paso (en un problema real se podría haber aplicado sobre todas las clases obtenidas hasta el momento). De esta forma obtenemos las siguientes clases de prueba (en este caso no hay nada que simplificar).

$$\begin{aligned} Extraer_1^{ME} &== [Extraer_3^{PE} \mid DEC_M2_ExtraerOk_Pred] \\ Extraer_2^{ME} &== [Extraer_3^{PE} \mid NDEC_M2_ExtraerOk_Pred] \\ Extraer_3^{ME} &== [Extraer_6^{PE} \mid DEC_M2_ExtraerOk_Pred] \\ Extraer_4^{ME} &== [Extraer_6^{PE} \mid NDEC_M2_ExtraerOk_Pred] \end{aligned}$$

donde $DEC_M2_ExtraerOk_Pred$ es el predicado del esquema $DEC_M2_ExtraerOk$ y $NDEC_M2_ExtraerOk_Pred$ es el de $\neg DEC_M2_ExtraerOk$ (hemos escrito los predicados de esta forma para ahorrar espacio; en la práctica deben escribirse los predicados explícitamente).

Observar que todas las clases obtenidas pueden ser refinadas nuevamente aplicando FND.

3.7. Causa-efecto (CE)

Esta es una táctica tradicional para testear un programa. Se trata de ver qué resultados (*efectos*) de un programa queremos asegurar que son correctos por lo que vamos a suministrarle las entradas (*causas*) necesarias para obtener esos resultados. De esta forma estamos probando que el programa funciona correctamente en los casos que más nos interesan. Por ejemplo, si el programa fuese el controlador de un ascensor querríamos estar seguros que la puerta se abre sólo si el ascensor está detenido en un piso. Aquí el efecto es "puerta cerrada" por lo que debemos rastrear todas las situaciones en que la puerta debería permanecer cerrada y testear el programa: si en alguno de los casos la puerta se abre, entonces hemos detectado un error importante; si en ninguno de esos casos la puerta se abre entonces se abrirá cuando no hay peligro.

La aplicación de la táctica comienza definiendo una partición del OS cuyas clases de prueba capturan los efectos que nos interesan. Luego se determina la partición del VIS tal que cada una

⁷Tener en cuenta la semántica Z de un esquema.

de sus clases produce uno de esos efectos. Finalmente, estos últimos predicados se utilizan para particionar una o más clases de prueba del último nivel del árbol de pruebas.

Para este ejemplo nos interesa ver si el programa funciona correctamente cuando el saldo de la caja de ahorros queda en cero, cuando queda entre cero y *saldoMin* y cuando queda por encima de aquel⁸. Estos efectos se representan de la siguiente forma:

$$\begin{aligned}
\text{SaldoMenor_cero} &== [\Delta \text{CajaAhorrosBanco}; d? : \text{DNI} \mid d? \in \text{dom } ca \wedge ca' d? < 0] \\
\text{SaldoMenor_saldoMin} &== \\
&[\Delta \text{CajaAhorrosBanco}; d? : \text{DNI} \mid d? \in \text{dom } ca \wedge 0 \leq ca' d? \wedge ca' d? < \text{saldoMin}] \\
\text{SaldoMayor_saldoMin} &== \\
&[\Delta \text{CajaAhorrosBanco}; d? : \text{DNI} \mid d? \in \text{dom } ca \wedge \text{saldoMin} \leq ca' d?]
\end{aligned}$$

En tanto que las causas (o clases de prueba del *VIS*) que producen cada uno de esos efectos son:

$$\begin{aligned}
\text{Causa_SaldoMenor_cero} &== [\text{VIS}_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge ca' d? - m? < 0] \\
\text{Causa_SaldoMenor_saldoMin} &== \\
&[\text{VIS}_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge 0 \leq ca' d? \wedge ca' d? - m? \leq \text{saldoMin}] \\
\text{Causa_SaldoMayor_saldoMin} &== [\text{VIS}_{\text{Extraer}} \mid d? \in \text{dom } ca \wedge \text{saldoMin} \leq ca' d? - m?]
\end{aligned}$$

Por lo tanto, solo queda combinar las causas con todas o algunas de las clases de prueba del último nivel del árbol. En este ejemplo decidimos hacerlo sobre las clases Extraer_i^{ME} con i de 1 a 4, lo que nos da las siguientes clases de prueba (tampoco aquí es necesario podar esta parte del árbol, aunque hemos omitido la cláusula $d? \in \text{dom } ca$ pues ya está incluida en las clases de prueba Extraer_i^{ME} para todo i).

$$\begin{aligned}
\text{Extraer}_1^{CE} &== [\text{Extraer}_1^{ME} \mid ca' d? - m? < 0] \\
\text{Extraer}_2^{CE} &== [\text{Extraer}_1^{ME} \mid 0 \leq ca' d? \wedge ca' d? - m? \leq \text{saldoMin}] \\
\text{Extraer}_3^{CE} &== [\text{Extraer}_1^{ME} \mid \text{saldoMin} \leq ca' d? - m?] \\
\text{Extraer}_4^{CE} &== [\text{Extraer}_2^{ME} \mid ca' d? - m? < 0] \\
\text{Extraer}_5^{CE} &== [\text{Extraer}_2^{ME} \mid 0 \leq ca' d? \wedge ca' d? - m? \leq \text{saldoMin}] \\
\text{Extraer}_6^{CE} &== [\text{Extraer}_2^{ME} \mid \text{saldoMin} \leq ca' d? - m?] \\
\text{Extraer}_7^{CE} &== [\text{Extraer}_3^{ME} \mid ca' d? - m? < 0] \\
\text{Extraer}_8^{CE} &== [\text{Extraer}_3^{ME} \mid 0 \leq ca' d? \wedge ca' d? - m? \leq \text{saldoMin}] \\
\text{Extraer}_9^{CE} &== [\text{Extraer}_3^{ME} \mid \text{saldoMin} \leq ca' d? - m?] \\
\text{Extraer}_{10}^{CE} &== [\text{Extraer}_4^{ME} \mid ca' d? - m? < 0] \\
\text{Extraer}_{11}^{CE} &== [\text{Extraer}_4^{ME} \mid 0 \leq ca' d? \wedge ca' d? - m? \leq \text{saldoMin}] \\
\text{Extraer}_{12}^{CE} &== [\text{Extraer}_4^{ME} \mid \text{saldoMin} \leq ca' d? - m?]
\end{aligned}$$

3.8. Otras tácticas

Los testers pueden definir todas las tácticas que deseen tratando de lograr que siempre generen particiones. Las tácticas que mostramos en este apunte no son todas las que existen aunque son las más generales y habituales. Entre los ejercicios de práctica se le pide al alumno que desarrolle una táctica que tenga en cuenta las definiciones axiomáticas de una especificación. Por otra parte, a continuación, introducimos muy brevemente y sin aplicarlas a *Extraer* otras tácticas que pueden tenerse en cuenta en algunos casos específicos.

⁸Si bien hay otros efectos interesantes, sus causas ya están incluidas en las clases de prueba que tenemos hasta el momento por lo que no aportarían nada significativo.

3.8.1. Tipos Libres (TL)

Esta táctica apunta a tener en cuenta todas las formas posibles de construir un término que tiene un tipo libre. Los tipos libres en \mathbb{Z} se utilizan tanto para definir tipos enumerados (una forma trivial de tipo inductivo) y tipos inductivos. Entonces, si en una especificación una de las variables del VIS tiene tipo libre, la táctica sugiere particionar el VIS en tantas clases como formas haya de construir un valor para esa variable.

Por ejemplo, la siguiente definición define un tipo que une los caracteres con un código de error especial.

$$[CHAR]$$

$$CHANNEL_CONTENT ::= error \mid msg \langle\langle CHAR \rangle\rangle$$

Entonces si el VIS de una cierta operación $Send$ incluye una variable de tipo $CHANNEL_CONTENT$:

$$VIS_{Send} == [lchannel : CHANNEL_CONTENT; \dots]$$

la táctica TL prescribe la siguiente partición:

$$Send_1^{TL} == [VIS_{Send} \mid lchannel = error]$$

$$Send_2^{TL} == [VIS_{Send} \mid \exists c : CHAR \bullet lchannel = msg \ c]$$

3.8.2. Límites de Implementación (LI)

Si bien en las especificaciones se trata de utilizar los tipos más abstractos posibles como \mathbb{Z} , las implementaciones se hacen en base a subconjuntos finitos de dichos tipos. En consecuencia los programadores, al convertir la especificación en implementación, deben incluir código para resolver esa diferencia pues caso contrario el programa funcionará incorrectamente y de manera inesperada.

Por lo tanto proponemos una táctica que tenga en cuenta la existencia de límites en las variables de implementación generando una partición del VIS que considere valores por debajo, iguales y por encima de esos límites.

Por ejemplo, si el VIS de cierta operación $Suma$ contiene una variable de tipo \mathbb{Z} :

$$VIS_{Suma} == [cant : \mathbb{Z}; \dots]$$

y sabemos que el programa está implementado en C sobre Linux y que $cant$ tiene tipo `int`, entonces la táctica LI establece la siguiente partición⁹:

$$Suma_1^{LI} == [VIS_{Suma} \mid cant < -2147483648]$$

$$Suma_2^{LI} == [VIS_{Suma} \mid cant = -2147483648]$$

$$Suma_3^{LI} == [VIS_{Suma} \mid -2147483648 < cant \wedge cant < 2147483647]$$

$$Suma_4^{LI} == [VIS_{Suma} \mid cant = 2147483647]$$

$$Suma_5^{LI} == [VIS_{Suma} \mid 2147483647 < cant]$$

Al contrario de lo que sucede con la mayor parte de los casos de prueba, si algún representante de las clases 1 o 5 es exitoso puede ocurrir que el equipo de desarrollo decida no modificar la implementación.

En la mayoría de los casos la finitud de recursos de una computadora real no debe ser motivo para escribir una especificación que mencione tales límites. Por ejemplo, si en la especificación la variable A tiene tipo $\mathbb{P}X$ y se la implementa con una lista enlazada, en teoría estaríamos cometiendo un error pues la lista tendrá un límite dado por la cantidad de memoria de la máquina o del proceso.

⁹En este caso tomamos los límites de `int` en la `libc` de Linux 2.6.20.15.

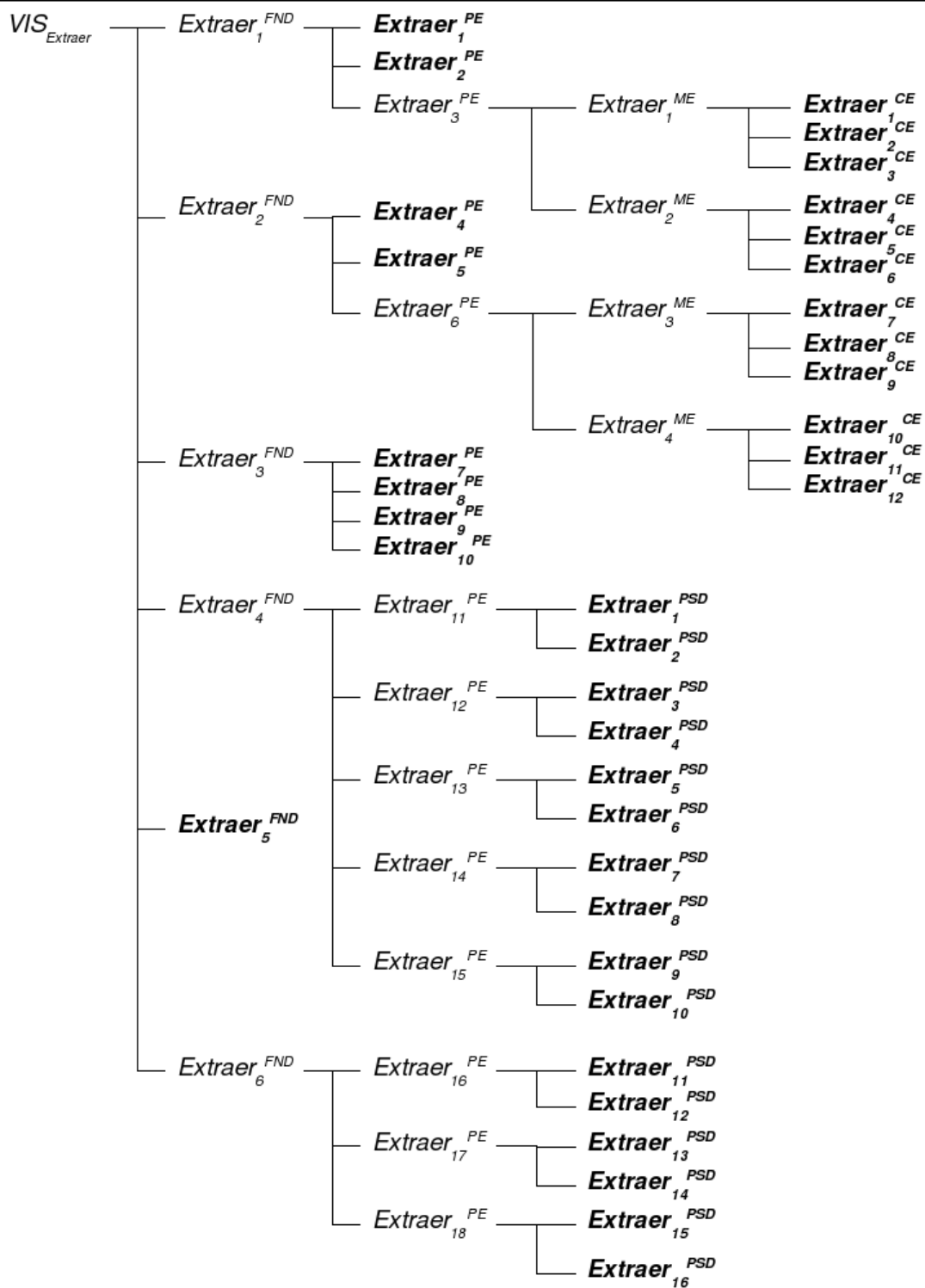


Figura 11: Las clases de prueba en negrita constituyen el último nivel del árbol, desde el cual se sacan los casos de prueba abstractos. Hay 37 clases de prueba en el último nivel.

$$\left| \begin{array}{l} char_a, char_b : CHAR \\ dni_1, dni_2, dni_3 : DNI \end{array} \right.$$

Una vez que contamos con las constantes (en realidad habrá que ir definiéndolas a medida que generemos casos de prueba) podemos definir los casos de prueba. Como los casos de prueba deben pertenecer a una clase de prueba usaremos una notación semejante a la que usamos hasta el momento. Por ejemplo, los siguientes son algunos de los casos de prueba para *Extraer* correspondientes a tres clases de prueba del árbol de la Figura 11.

$$\begin{aligned} Extraer_1^{CP} &== [Extraer_1^{PE} \mid ca = \{dni_1 \mapsto 100\} \wedge d? = dni_1 \wedge m? = 10] \\ Extraer_2^{CP} &== [Extraer_2^{PE} \mid ca = \{dni_1 \mapsto 100, dni_2 \mapsto 100\} \wedge d? = dni_1 \wedge m? = 10] \\ Extraer_3^{CP} &== \\ & [Extraer_1^{CE} \mid ca = \{dni_1 \mapsto -100, dni_2 \mapsto 100, dni_3 \mapsto 100\} \wedge d? = dni_1 \wedge m? = 10] \end{aligned}$$

Observar que en todos los casos se trata de igualdades para cada una de las variables del *VIS* y que la clase de prueba incluida en cada uno de los esquemas $Extraer_i^{CP}$ con $i : 1, 2, 3$ es una de las clases de prueba del último nivel del árbol de pruebas de *Extraer*. Además, las constantes seleccionadas deben verificar todas las condiciones de la clase de prueba incluida pues de lo contrario el esquema estaría vacío y en realidad no tendríamos un caso de prueba o probaríamos el programa con una entrada que no corresponde con la alternativa funcional que queremos capturar.

Si quisiéramos definir más de un caso de prueba por clase de prueba simplemente debemos escribir, por ejemplo:

$$Extraer_4^{CP} == [Extraer_1^{PE} \mid ca = \{dni_1 \mapsto 500\} \wedge d? = dni_1 \wedge m? = 100]$$

donde repetimos $Extraer_1^{PE}$ y seleccionamos diferentes constantes que en $Extraer_1^{CP}$.

Una herramienta que asista al ingeniero en la automatización de esta forma de testing podría prestar ayuda en dos niveles para la generación de los casos de prueba:

1. Corroborar que los casos propuestos por el ingeniero pertenecen a las clases de prueba indicadas. Esto es totalmente automatizable.
2. Generar casos de prueba que pertenezcan a cada clase de prueba. Si bien esto no es completamente automatizable se puede recurrir a modelos finitos y resolución automática de restricciones que podrían reducir considerablemente el trabajo manual.

Referencias

- [CAR10] Maximiliano Cristiá, Pablo Albertengo, and Pablo Rodríguez Monetti. Pruning testing trees in the test template framework by detecting mathematical contradictions. In *SEFM*, 2010.
- [CR09] 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.
- [Cri05] Maximiliano Cristiá. Especificación Z de parte de los sistemas de un banco. <http://www.fceia.unr.edu.ar/asist/a-z.pdf>, 2005.
- [Fit96] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

- [HP95] Hans Martin Hörcher and Jan Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4:309–327, 1995.
- [PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.
- [SC96] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
- [SMFM00] S. Souza, J. Maldonado, S. Fabbri, and P. Masiero. Statecharts Specifications: A Family of Coverage Testing Criteria. In *CLEI'2000 - XXVI Latin-American Conference of Informatics*. CLEI, 2000.
- [Sto93] P. Stocks. *Applying Formal Methods to Software Testing*. PhD thesis, Department of Computer Science, University of Queensland, 1993.