Apunte de clase

Test Template Framework

Una introducción al testing basado en modelos

Maximiliano Cristiá

Ingeniería de Software 2
Licenciatura en Ciencias de la Computación
Universidad Nacional de Rosario

Índice

1. MBT y el Test Template Framework

Como ya mencionamos, el testing basado en modelos[1] (MBT) propone testear un programa (P) con casos de prueba generados a partir del análisis de su especificación formal (S) [14, 10]

En este apunte veremos con cierto detalle un método específico de MBT llamado *Test Template Framework* (TTF). El TTF fue propuesto por Phil Stocks y David Carrington a mediados de los noventa [13]. Alrededor de 2009 se implementó FASTEST, la primera herramienta en automatizar la generación de casos de prueba basada en el TTF [5, 2, 1]. Más recientemente {*log*} incluye una implementación del TTF totalmente integrada con esa herramienta lo que redunda en mayor eficiencia en la generación automática de casos de prueba partiendo de un forgrama {*log*} [3].

1.1. El proceso de MBT

La Figura 1 muestra las etapas básicas del proceso de testing basado en modelos. Como podemos ver, el proceso comienza analizando S para GENERAR *casos de prueba abstractos* (ATC). Un ATC[2] es abstracto porque asocia valores constantes a las variables de entrada de S. Es decir un ATC es un objeto que vive en el nivel (abstracto) de S. El segundo paso del proceso consiste en REFINAR los ATC en *casos de prueba concretos* (CTC)[3]. Un CTC[4] es un caso de prueba como los que hemos visto en testing estructural. Es decir, un CTC asocia valores constantes a las variables de entrada de P. El tercer paso radica en EJECUTAR P tomando como entrada cada uno de los CTC lo que produce alguna salida o efecto en el entorno del programa. El paso COMPROBAR incluye dos etapas: en la primera, llamada ABSTRAER, se intenta abstraer las salidas producidas por P para cada CTC; y en la segunda, llamada SUSTITUIR, cada ATC y la salida abstracta correspondiente se sustituyen en S. Al efectuar esta sustitución S se vuelve una fórmula cerrada (es decir una fórmula sin variables libres), que denotamos con $\hat{S}$. Si la salida producida por P no se puede abstraer o si $\hat{S}$ no reduce a *true*, entonces hay un error en P; caso contrario el ATC no detectó ningún error. La salida producida por P se llama *salida real* mientras que la salida especificada por S para un cierto ATC se denomina *salida esperada*.

Para ejemplificar el proceso de la Figura 1 consideremos una especificación que tiene la variable de estado *set* de tipo $\mathbb{P}(\mathbb{Z})$. Un ATC para esa especificación podría ser asociar *set* con el valor $\{1, 5, -3\}$. Ahora supongamos que la especificación se implementó como un programa Java donde la variable *set* se implementó como una lista de nombre `set`. Entonces el CTC correspondiente al ATC anterior es[5] `set.add(1).add(5).add(-3)`. Analicemos las siguientes situaciones:

- Supongamos que al ejecutar P con ese CTC el programa termina con una excepción del tipo *puntero nulo*. Entonces es imposible abstraer esa salida al nivel de la especificación por lo que se ha detectado un error en P.
- Ahora supongamos que al ejecutar P con ese CTC el programa termina con `set` igual a `set.add(1).add(-3).add(-3)`—es decir, este valor de `set` corresponde al valor de *set* en el estado de llegada, o sea es una salida del programa. Este valor de `set` no se puede abstraer porque no es un conjunto (se repite -3), por lo que se ha detectado un error en P.

[1] 'model-based testing', en inglés.
[2] 'abstract test case', en inglés.
[3] Este paso también se puede llamar *concretizar* [14, Capítulo 8] o *reificar* [13].
[4] 'concrete test case', en inglés.
[5] Suponemos que `set` acaba de ser inicializada con el conjunto vacío.

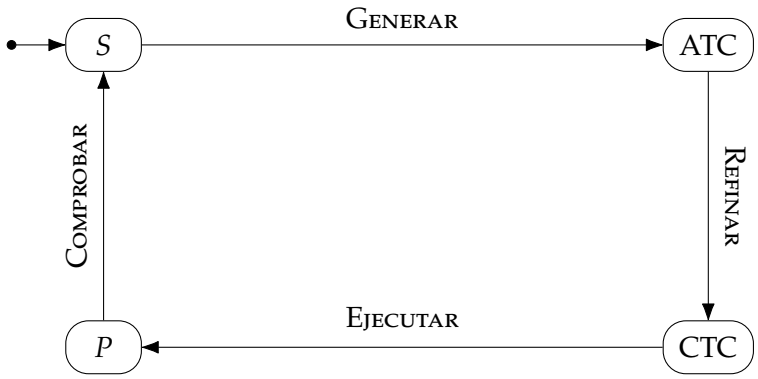

Figura 1: El proceso de testing basado en modelos

- Una tercera posibilidad es que el resultado de P fuese `set.add(1).add(5).add(-2)` que sí se puede abstraer a *set* = $\{1,5,-2\}$. Pero supongamos que S establece que la salida esperada del ATC *set* = $\{1,5,-3\}$ debe ser *set* = $\{1,5,-4\}$. Entonces habremos encontrado un error en P puesto que la salida real no corresponde a la salida esperada.
- Finalmente, si P termina con `set.add(1).add(5).add(-4)`, el ATC no habrá detectado un error en P (lo que no significa que no haya errores en P).

El paso Generar es el más importante puesto que es aquí donde se decide con cuántos y con cuáles casos de prueba se va a testear el programa. Este paso determina el nivel de cobertura que tendrá la campaña de testing. Todos los métodos de MBT apuntan a que el paso Generar sea el resultado de un enfoque sistemático, disciplinado y cuantificable. Tan es así que en varios métodos de MBT el paso Generar se reduce a la ejecución de un algoritmo que determina cuántos y cuáles casos de prueba se van a generar. En general se espera que no sea el desarrollador quien seleccione los casos de prueba sino que se limite a establecer la estrategia general de la campaña de testing.

Si bien la Figura 1 representa en términos generales el proceso de testing de cualquier método de MBT, existen métodos que no adhieren exactamente a ese proceso. En efecto, algunos métodos de MBT consideran que un ATC no solo debe asociar valores a las variables de entrada sino que también debe asociar las salidas esperadas con las variables de salida. Esto significa que cuando se refina un ATC también se refinan las salidas esperadas. En este caso el paso Comprobar solo compara las salidas reales con las salidas esperadas refinadas. Nosotros adherimos a la Figura 1 porque cuando S se basa en la teoría de conjuntos (como B y {*log*}) la comparación de salidas reales con salidas esperadas no es trivial. Volvamos sobre el ejemplo de la especificación con la variable *set*. Supongamos que la salida esperada (para un cierto ATC) es $\{1,5,-4\}$ la cual se refina a `set.add(1).add(5).add(-4)` (recordar que `set` es una lista, no un conjunto, a nivel del programa). Ahora supongamos que P termina con `set.add(-4).add(1).add(5)`. Observen que la salida real y la esperada representan el mismo conjunto pero son listas distintas. Si la comparación de las salidas se hace a nivel de P parecería que se ha encontrado un error, cuando en realidad no es el caso. Por lo tanto, no se pueden comparar salidas a nivel de P por lo que resulta inevitable abstraer las salidas reales. Si es inevitable abstraer las salidas reales entonces lo más eficiente es no calcular las salidas esperadas cuando se genera el ATC sino que resulta más conveniente ejecutar la subetapa Sustituir la cual puede echar mano, por ejemplo,

de la capacidad de {*log*} para resolver fórmulas de la teoría de conjuntos.

1.2. Hacia una formalización del proceso de MBT

De ahora en más consideramos que S es una operación (i.e. una transición de estados) definida sobre alguna máquina de estados (por ejemplo de {*log*}). Más aun, consideramos que S es un predicado que depende de variables de estado (de partida y de llegada) y de variables de entrada y de salida. Supongamos entonces que nuestra especificación es el siguiente predicado:

$$q(s, i, s', o) \mathrel{\widehat{=}} \text{alguna fórmula que depende de } s,\ i,\ s' \text{ y } o \tag{1}$$

donde s representa una variable de estado de partida, s' la correspondiente variable de estado de llegada[6], i es una variable de entrada, y o es una variable de salida. En general diremos que s e i son las variables de entrada de q (o sea sin distinguir entre variables de estado y variables de entrada propiamente dichas); mientras que s' y o son las variables de salida.

En consecuencia un ATC para q puede simbolizarse de la siguiente forma:

$$(s, i) \rightarrowtail (c_s, c_i) \tag{2}$$

donde c_s es una constante del tipo de s y c_i es una constante del tipo de i—por ejemplo, $set \rightarrowtail \{1, 5, -3\}$.

A continuación definimos $\mathcal{R}$ como la función que transforma cada ATC en un CTC. Es decir que $\mathcal{R}$ es una implementación del paso Refinar de la Figura 1. De forma similar definimos $\mathcal{A}$ como la función parcial que transforma salidas reales en salidas abstractas. O sea que $\mathcal{A}$ es una implementación de la subetapa Abstraer del paso Comprobar de la Figura 1. $\mathcal{A}$ es una función parcial porque no todas las salidas producidas por P se van a poder abstraer—como mostramos en el ejemplo anterior.

Entonces, la implementación P de q tiene un error en el CTC $\mathcal{R}((s, i) \rightarrowtail (c_s, c_i))$ si y solo si[7]:

$$(P \circ \mathcal{R})((s, i) \rightarrowtail (c_s, c_i)) \notin \text{dom}(\mathcal{A}) \vee \neg\, q(c_s, c_i, (\mathcal{A} \circ P \circ \mathcal{R})((s, i) \rightarrowtail (c_s, c_i)))$$

Claramente el primer término de la disyunción es la formalización de la idea de que ciertas salidas reales no se pueden abstraer lo que significa que se ha detectado un error. En tanto el segundo término de la disyunción corresponde a la sustitución en q del ATC y la correspondiente salida abstracta formalizada como $(\mathcal{A} \circ P \circ \mathcal{R})((s, i) \rightarrowtail (c_s, c_i))$.

Existe una posibilidad más que indica que hay errores en P que aun no hemos mencionado. Si las funciones $\mathcal{R}$ y $\mathcal{A}$ ni siquiera se pueden definir entonces P es erróneo. Por ejemplo, imaginemos que el desarrollador olvidó definir una variable en P para la variable de estado *set*. En ese caso $\mathcal{R}$ no se puede ni siquiera definir y en consecuencia inmediatamente determinamos que P es erróneo. Al inicio no consideramos esta situación porque en este caso P casi ni sería una implementación de S. En otras palabras, P estaría tan mal que sería difícil argumentar que es una implementación de S.

[6] En {*log*} usamos $s\_$ en lugar de s' pero esta última convención es la más utilizada en varios lenguajes de especificación.

[7] $(f \circ g)$ denota composición funcional.

1.3. Automatización del proceso de MBT

El proceso de la Figura 1 se puede automatizar en muchísimos casos. El paso más complejo de automatizar (por mucho) es Generar. De hecho, si S es una fórmula que pertenece a algún fragmento no-decidible de una cierta teoría, la automatización del paso Generar podría no ser posible. El nudo del problema de la automatización de Generar radica en que en algún momento va a ser necesario determinar si una fórmula derivada de S es satisfacible o no, y en el primer caso se debe generar un *testigo* o solución. Las fórmulas que se generan durante el paso Generar, en general, van a pertenecer al mismo fragmento que S. Por lo tanto, si S pertenece a algún fragmento no-decidible es probable que sea imposible determinar de forma automática si las fórmulas generadas son satisfacibles o no.

La automatización de los otros pasos del proceso de la Figura 1 en general es más simple y requiere más que nada el desarrollo o la integración de herramientas [14, Capítulo 8] [4, 15]

¿Qué es el testing automático en la industria de software? Si bien MBT se usa en algunos proyectos de algunos sectores de la industria de software [11, 9], la grandísima mayoría de las empresas usa herramientas o *frameworks* de 'automatización' de testing del tipo xUnit. Estas herramientas solo automatizan el paso Ejecutar y una parte trivial del paso Comprobar. Como estas herramientas trabajan a nivel del programa el paso Refinar y la subetapa Abstraer no son necesarios. El paso Generar (por mucho el más complejo y crítico) lo debe ejecutar el desarrollador manualmente. Esto hace que sean los desarrolladores quienes seleccionan los casos de prueba de acuerdo a su parecer, por lo general sin aplicar ningún método sistemático que tenga alguna relación cuantificable con el grado de cobertura. Los desarrolladores también deben ejecutar manualmente la parte más compleja y crítica del paso Comprobar que consiste en calcular, de alguna forma, la salida esperada para cada CTC. Una vez calculadas las salidas esperadas las herramientas tipo xUnit se limitan a Ejecutar los CTC, comparar las salidas reales con las esperadas y emitir el veredicto final.

1.4. El paso Generar en el TTF

Si bien en la sección anterior consideramos todo el proceso de MBT, como ya dijimos, el paso más complejo y crítico (por mucho) es Generar. En general este paso es muy dependiente del método de MBT específico que se esté estudiando. En este curso veremos en detalle solo este paso y lo haremos siguiendo el TTF.

Generar, en general, se puede dividir en tres tareas como muestra la Figura 2:

1. Particionar. Consiste en particionar[8] el espacio de entrada de S en subconjuntos llamados *clases*, *especificaciones* o *condiciones* de test o de prueba. El espacio de entrada de S es el conjunto de todos los valores posibles que pueden tomar las variables de entrada de S.
2. Podar. Por cuestiones que veremos un poco más adelante, muchas clases de prueba son conjuntos vacíos por lo que deben ser eliminadas. Una vez eliminadas las clases vacías las restantes se pueden volver a particionar ejecutando la tarea Particionar nuevamente.
3. Grounding. Cuando hemos terminado con las tareas anteriores debemos generar un testigo para cada clase de prueba. Cada testigo es un ATC.

[8] La familia de conjuntos $\{A_i\}_{i\in I}$ es una partición del conjunto A si $A = \bigcup_{i\in I} A_i$ y $A_i \cap A_j = \emptyset$ para todo $i, j \in I$ y $i \neq j$.

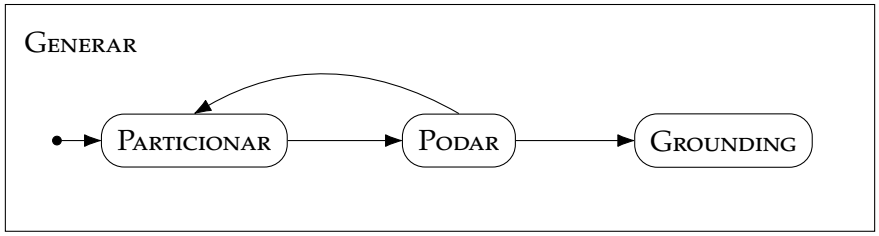

Figura 2: GENERAR en el TTF

Consideremos nuevamente que S está dada por un predicado de la forma:

$$q(s, i, s', o)$$

donde s, i, s' y o son como se indicó en (1) y además el tipo de s es T_s y el tipo de i es T_i. En este contexto el espacio de entrada de q, simbolizado con q_{IS}, se define como:

$$q_{IS} \mathrel{\widehat{=}} T_s \times T_i$$

Si PRE_q es la precondición de q entonces definimos el *espacio válido de entrada* (VIS) de q de la siguiente forma[9]:

$$q_{VIS} \mathrel{\widehat{=}} \{(x, y) \in q_{IS} \mid \text{PRE}_q(x, y)\}$$

1.4.1. Particionar, tácticas de testing y árbol de testing

Durante la etapa PARTICIONAR se particiona el VIS aplicando una o más *tácticas de testing*[10]. Una táctica de testing es una regla o procedimiento que especifica de qué manera se debe particionar el VIS o una clase de prueba. Cada táctica de testing indica cuántos subconjuntos tendrá la partición y cada subconjunto queda caracterizado por un predicado que depende de las variables de entrada de q. Las tácticas de testing permiten generar clases de prueba cuya intención es testear el programa de diferentes formas.

De esta forma si T es una táctica de testing donde $p_1^{\mathsf{T}}, \ldots, p_n^{\mathsf{T}}$ son los predicados que caracterizan a los subconjuntos especificados por T entonces la partición de q_{VIS} se describe dando las clases de prueba, q_i^{T}, así:

$$q_1^{\mathsf{T}} \mathrel{\widehat{=}} \{(x, y) \in q_{VIS} \mid p_1^{\mathsf{T}}(x, y)\}$$
$$q_2^{\mathsf{T}} \mathrel{\widehat{=}} \{(x, y) \in q_{VIS} \mid p_2^{\mathsf{T}}(x, y)\}$$
$$\vdots$$
$$q_n^{\mathsf{T}} \mathrel{\widehat{=}} \{(x, y) \in q_{VIS} \mid p_n^{\mathsf{T}}(x, y)\}$$

[9] Observen que PRE_q es un predicado que depende de las variables de estado anterior y de entrada. Revisar apunte sobre el método B.

[10] Stocks 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 se hace en el TTF. Por ejemplo, creemos que una estrategia de testing es el MBT o el testing estructural. Por este motivo preferimos táctica.

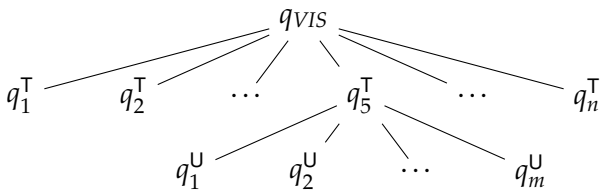

Figura 3: Árbol de testing de la operación q al aplicar la táctica T *a* q_{VIS} *y la táctica* U *a* q_5^{T}

Siendo que T es una táctica se espera que los conjuntos q_i^{T} verifiquen las siguientes propiedades:

$$q_{VIS} = q_1^{\mathsf{T}} \cup \cdots \cup q_n^{\mathsf{T}} \tag{3}$$

$$\forall i, j \in [1, n] : i \neq j \Rightarrow q_i^{\mathsf{T}} \cap q_j^{\mathsf{T}} = \emptyset \tag{4}$$

Es decir, se espera que la táctica de testing genere una partición. Si bien algunas tácticas no generan una partición todas verifican la propiedad (3) que garantiza cobertura. La pérdida de la propiedad (4) implica que el programa podría ser testeado más de una vez con el mismo caso de prueba.

Digamos que U es otra táctica cuyos predicados característicos son $p_1^{\mathsf{U}}, \ldots, p_m^{\mathsf{U}}$. Podemos aplicar U a una o más de las clases de prueba q_i^{T}. Si aplicamos U, por ejemplo, a la clase q_5^{T} las clases de prueba que se obtienen son las siguientes:

$$\begin{aligned} q_1^{\mathsf{U}} &\widehat{=} \{(x,y) \in q_5^{\mathsf{T}} \mid p_1^{\mathsf{U}}(x,y)\} \\ q_2^{\mathsf{U}} &\widehat{=} \{(x,y) \in q_5^{\mathsf{T}} \mid p_2^{\mathsf{U}}(x,y)\} \\ &\vdots \\ q_m^{\mathsf{U}} &\widehat{=} \{(x,y) \in q_5^{\mathsf{T}} \mid p_m^{\mathsf{U}}(x,y)\} \end{aligned} \tag{5}$$

Observen que ahora (x,y) es un elemento de q_5^{T} y no solo de q_{VIS}, en todas las clases de prueba. La táctica U debe cumplir las propiedades (3) y (4).

En el TTF las clases de prueba se organizan en un *árbol de testing* como se muestra en la Figura 3. La raíz del árbol es el VIS, el primer nivel de nodos son las clases de prueba obtenidas luego de aplicar la primera táctica, el segundo nivel son las clases de prueba obtenidas luego de particionar las clases del primer nivel y así sucesivamente.

Cuanto más profundo sea el árbol de testing y cuanto más diversas sean las tácticas que lo generan, mejor será la cobertura, mejor será el testing, mayores serán las chances de descubrir errores. En conflicto con esto está el hecho de que aplicar algunas tácticas a cualquier nodo del árbol de testing no genera necesariamente mejor testing. Si este es el caso, el testing no solo no será mejor sino que será menos eficiente porque se va a testear el programa con varios casos de prueba cuando se hubieran obtenido los mismos resultados con solo uno de ellos.

Criterio de aplicación de tácticas de testing. Para determinar si tiene sentido o no aplicar una táctica de testing a un nodo del árbol se debe tener en cuenta, al menos, lo siguiente:

- El objetivo es testear ciertas líneas de código de P cuya especificación es el predicado t.

- Estas líneas de código se ejecutan solo si las variables de entrada de P satisfacen ciertas restricciones que simbolizaremos con r. Es decir, $r \Rightarrow t$. Sea $var(r,t)$ el conjunto de variables de entrada que aparecen en r o en t.
- Sea p el predicado que caracteriza la clase de prueba que queremos particionar.
- Sean $s_1,\ldots,s_n$ los predicados que caracterizan la táctica que queremos aplicar. Sea $var(s_1,\ldots,s_n)$ el conjunto de variables de entrada que aparecen en algún s_i.
- Si $p \not\Rightarrow r$ o si $var(s_1,\ldots,s_n) \cap var(r,t) = \emptyset$ entonces no tiene sentido aplicar esa táctica sobre ese nodo del árbol.

Como ejemplo consideremos un programa P que tiene dos variables de entrada, x e y, ambas de tipo $\mathbb{Z}$. Supongamos que la especificación de P es el predicado $q(x,y,o)$ donde o es una variable de salida. Tenemos como objetivo testear ciertas líneas de código de P cuya especificación t solo depende de la variable x. Esas líneas de código se ejecutan cuando $0 < x$. Es decir, tenemos $r \mathrel{\widehat{=}} 0 < x$ y $var(r,t) = \{x\}$. Entonces vale lo siguiente:

- Si el predicado de la clase de prueba que queremos particionar es $p \mathrel{\widehat{=}} x \leq 0$, entonces no tiene sentido hacer una partición dado nuestro objetivo porque $x \leq 0 \not\Rightarrow 0 < x$.
- Si la táctica que vamos a aplicar está dada por los predicados:

  $$s_1 \mathrel{\widehat{=}} y < 0;\ s_2 \mathrel{\widehat{=}} y = 0;\ s_3 \mathrel{\widehat{=}} 0 < y$$

  entonces no tiene sentido hacer una partición dado nuestro objetivo porque x no es una variable que aparezca en los s_i.
- Si el predicado de la clase de prueba que queremos particionar es $p \mathrel{\widehat{=}} 1 < x$ y si la táctica que vamos a aplicar está dada por los predicados:

  $$s_1 \mathrel{\widehat{=}} x \leq -3;\ s_2 \mathrel{\widehat{=}} -3 < x < 0;\ s_3 \mathrel{\widehat{=}} x = 0;\ s_4 \mathrel{\widehat{=}} 0 < x < 3;\ s_5 \mathrel{\widehat{=}} 3 \leq x \tag{6}$$

  entonces tiene sentido hacer una partición dado nuestro objetivo porque $p \Rightarrow r$ y x aparece en los s_i.

  Noten que luego de aplicar la nueva táctica tendríamos un caso de prueba donde $x = 2$ y otro donde $3 \leq x$ mientras que antes de aplicar esta táctica teníamos solo un caso de prueba dado por $1 < x$. Si bien el hecho de testear P con $x = 2$ y con, por caso, $x = 5$ no garantiza descubrir posibles errores, hay más chances de hacerlo que probando solo con $x = 2$. Al mismo tiempo, mirando P, podría ser evidente que testear P con $x = 5$ no va a agregar calidad al testing y que por lo tanto sería mejor testearlo solo con $x = 2$.

Por lo dicho en el último ítem, es que decimos que al momento de aplicar una táctica a un nodo se deben tener en cuenta *al menos* las condiciones enunciadas más arriba. En otras palabras, si las condiciones anteriores *no* se verifican el testing no será mejor y será menos eficiente; pero si se cumplen, no hay garantía de que el testing sea realmente mejor, aunque probablemente lo sea.

Automatización de Particionar. La decisión de qué táctica aplicar sobre cuál nodo del árbol se deja en manos del desarrollador. La generación de las clases de prueba y la actualización del árbol de testing es totalmente automática. De esta forma el desarrollador queda a cargo de la actividad más creativa que es determinar la estrategia de testing (i.e. qué testear y qué

no) mientras que la herramienta se encarga de la tarea más pesada y mecánica de generar las clases de prueba y organizarlas en el árbol. En {*log*}-TTF existen comandos de usuario que permiten aplicar diversas tácticas en nodos o subárboles indicados por el usuario. Estos comandos manipulan simbólicamente los predicados y gestionan el árbol de testing.

1.4.2. Podar el árbol de testing

Cuando el árbol de testing tiene algunos niveles es conveniente ejecutar la tarea Podar. Como ya mencionamos Podar elimina clases de prueba vacías del árbol de testing.

¿Por qué habría clases de prueba vacías en el árbol de testing? Consideremos la operación q que tiene una variable de entrada de tipo $\mathbb{Z}$. Supongamos que $\text{PRE}_q \mathrel{\widehat{=}} true$ por lo que $q_{VIS} = q_{IS} \mathrel{\widehat{=}} \mathbb{Z}$. Ahora tomamos la clase de prueba $q_1 \mathrel{\widehat{=}} \{x \in q_{VIS} \mid 1 < x\}$, surgida al aplicar una cierta táctica, y aplicamos otra táctica caracterizada por la partición dada en (6):

$$\begin{aligned} q_{11} &\mathrel{\widehat{=}} \{x \in q_1 \mid x \leq -3\} \\ q_{12} &\mathrel{\widehat{=}} \{x \in q_1 \mid -3 < x < 0\} \\ q_{13} &\mathrel{\widehat{=}} \{x \in q_1 \mid x = 0\} \\ q_{14} &\mathrel{\widehat{=}} \{x \in q_1 \mid 0 < x < 3\} \\ q_{15} &\mathrel{\widehat{=}} \{x \in q_1 \mid 3 \leq x\} \end{aligned} \tag{7}$$

Es fácil deducir que q_{11}, q_{12} y q_{13} son conjuntos vacíos puesto que los elementos de q_1 son números mayores a 1. En consecuencia esas clases de prueba deben ser eliminadas del árbol de testing puesto que no es posible extraer casos de prueba de ellas. Estas clases de prueba representan situaciones imposibles de testear en el programa.

Observen que decir que q_{11} es un conjunto vacío es equivalente a decir que el predicado que lo caracteriza es insatisfacible. En efecto, siendo $1 < x \wedge x \leq -3$ el predicado de q_{11} resulta evidente que es insatisfacible. Por eso hablamos de clases de prueba vacías o insatisfacibles.

Es conveniente ejecutar la tarea Podar cada vez que se aplica una táctica de testing sobre uno o más nodos del árbol de testing. De esta forma se evita generar árboles grandes donde muchas clases de prueba son insatisfacibles.

Automatización de Podar. La automatización de Podar depende de la decidibilidad de la teoría sobre la cual se escribe S. En efecto, dado que para podar una clase de prueba hay que determinar su satisfacibilidad, la posibilidad de automatizar esta actividad depende de la existencia de un algoritmo capaz de determinar la satisfacibilidad de fórmulas de una cierta teoría en un tiempo finito. Como ya sabemos, la existencia de tales algoritmos (llamados *procedimientos de decisión*) para una cierta teoría es la definición del concepto de decidibilidad.

En nuestro caso la especificación S se escribe en términos de lógica de primer orden, teoría de conjuntos y aritmética entera. Ninguna de estas tres teorías es decidible. Por lo tanto, en general, Podar no se puede automatizar para las especificaciones con las que nos interesa trabajar. Sin embargo, existen fragmentos muy expresivos de las tres teorías que sí son decidibles. Por ejemplo, la aritmética lineal entera es decidible. {*log*} implementa procedimientos de decisión para varios de los fragmentos decidibles de las teorías en cuestión (e.g. [8, 6]). Por lo tanto {*log*} está en condiciones de automatizar gran parte de Podar para una clase muy grande de especificaciones que se dan en problemas prácticos.

1.4.3. Grounding: generación de casos de prueba abstractos

Grounding es el proceso por medio del cual se generan los ATC. En el TTF un ATC asocia las variables de entrada de la especificación con constantes, como definimos en (2). El vector de constantes de un ATC es un elemento que pertenece a una de las hojas del árbol de testing. Es decir, en el TTF los ATC se extraen solo de las hojas del árbol. Por ejemplo, un ATC de la clase q_{15} dada en (7) es $x \leftarrow\!\shortmid 5$. En lógica una *expresión ground* es un término que no contiene variables. Nosotros llamamos Grounding al proceso de encontrar expresiones *ground* que pertenezcan a clases de prueba; estas expresiones también fungen como soluciones o testigos de fórmulas.

Encontrar una constante que pertenezca a un conjunto es equivalente a encontrar una constante que satisfaga el predicado que caracteriza a ese conjunto:

$$c \in \{x \mid \phi(x)\} \Leftrightarrow \phi(c)$$

Por lo tanto, en el TTF la generación de ATC implica resolver un problema de satisfacibilidad. Este problema de satisfacibilidad es el mismo que hay que resolver cuando se ejecuta Podar. Es decir, si una clase de prueba es hoja de un árbol de testing se debe determinar si es satisfacible, en cuyo caso se debe proveer una solución que será el caso de prueba; o si no lo es, en cuyo caso se la elimina del árbol.

Para comprender por qué solo se usan las hojas del árbol consideremos las clases de prueba $q_i^{\cup}$ especificadas en (5). Como podemos ver en el árbol de la Figura 3, esas clases de prueba son hojas. Consideremos, por ejemplo, la hoja $q_1^{\cup}$. Un elemento de esta clase es un par ordenado (a,b) tal que pertenece a $q_1^{\cup}$. Si $(a,b) \in q_1^{\cup}$ entonces también vale $(a,b) \in q_5^{\intercal}$, por la misma definición de $q_1^{\cup}$. Por otro lado, como $(a,b) \in q_5^{\intercal}$ solo si $(a,b) \in q_{VIS}$ y vale $p_5^{\intercal}(a,b)$, entonces podemos escribir las clases $q_i^{\cup}$ de la siguiente forma:

$$\begin{aligned}
q_1^{\cup} &\widehat{=} \{(x,y) \in q_{VIS} \mid p_5^{\intercal}(x,y) \wedge p_1^{\cup}(x,y)\} \\
q_2^{\cup} &\widehat{=} \{(x,y) \in q_{VIS} \mid p_5^{\intercal}(x,y) \wedge p_2^{\cup}(x,y)\} \\
&\vdots \\
q_m^{\cup} &\widehat{=} \{(x,y) \in q_{VIS} \mid p_5^{\intercal}(x,y) \wedge p_m^{\cup}(x,y)\}
\end{aligned}$$

O sea que sustituimos $q_5^{\intercal}$ con su definición en cada clase de prueba. De la misma forma podemos sustituir q_{VIS} obteniendo:

$$\begin{aligned}
q_1^{\cup} &\widehat{=} \{(x,y) \in q_{IS} \mid \textsc{pre}_q(x,y) \wedge p_5^{\intercal}(x,y) \wedge p_1^{\cup}(x,y)\} \\
q_2^{\cup} &\widehat{=} \{(x,y) \in q_{IS} \mid \textsc{pre}_q(x,y) \wedge p_5^{\intercal}(x,y) \wedge p_2^{\cup}(x,y)\} \\
&\vdots \\
q_m^{\cup} &\widehat{=} \{(x,y) \in q_{IS} \mid \textsc{pre}_q(x,y) \wedge p_5^{\intercal}(x,y) \wedge p_m^{\cup}(x,y)\}
\end{aligned} \tag{8}$$

Como q_{IS} es simplemente un producto cartesiano no agrega ningún predicado a la definición de las clases de prueba y por consiguiente no lo sustituimos.

Esto quiere decir que si tomamos un elemento que pertenezca a $q_1^{\cup}$ será un elemento que satisfará $\textsc{pre}_q(x,y) \wedge p_5^{\intercal}(x,y) \wedge p_1^{\cup}(x,y)$. Es decir que si tomamos un elemento que pertenezca a $q_1^{\cup}$ también será un elemento de $q_5^{\intercal}$. En consecuencia, si tomamos un elemento que pertenece a una

hoja es también un elemento que pertenece a todos los nodos del camino que une esa hoja con la raíz del árbol de testing. Por este motivo en el TTF solo se generan ATC tomando elementos de las hojas del árbol.

En términos de testing, si testeamos el programa con un ATC derivado de q_1^{U} también estamos testeando el programa con un ATC derivado del nodo padre q_5^{T}—y en general de todos los nodos que forman el camino desde la hoja hasta la raíz. No obstante, la inversa de la afirmación anterior no es cierta: si testeamos el programa con un ATC derivado de q_5^{T} no necesariamente estamos testeando el programa con un ATC derivado de q_1^{U}.

Este análisis nos ayuda a ver por qué cuanto más profundo sea el árbol de testing mejor será el testing. Claramente, cada nivel del árbol conjuga un predicado más a las clases del nivel anterior. Esto se traduce en ATC que testean condiciones cada vez más precisas, más exigentes, más específicas.

Además de derivar ATC solo de las hojas, el TTF establece que se debe generar un *único* ATC por hoja del árbol de testing. Esto se debe a que cada clase de prueba que es hoja se considera que es una *clase de equivalencia*. Es de equivalencia porque se supone que el programa se va comportar igual para todos los elementos de la clase de prueba. Es decir, cualquier elemento de la clase de prueba va a descubrir el mismo error en P o no lo hará ninguno. Si existen motivos para suponer que P no se va a comportar igual para todos los elementos de una clase de prueba entonces se la debe particionar aplicando una nueva táctica, en lugar de derivar más de un ATC sin otro criterio que "uno solo me parece poco".

Veamos un ejemplo sobre las clases de prueba como clases de equivalencia. Digamos que P está implementado en lenguaje C y tiene la variable de entrada `num` de tipo `int` con signo que en la mayoría de los sistemas modernos almacena valores en el intervalo $[-(2^{31}), 2^{31}-1]$. La especificación S usa la variable de entrada *num* de tipo $\mathbb{Z}$. Además sabemos que P ejecuta un ciclo de 1 a `num` si este es mayor o igual a 1. S considera las precondiciones $num < 1$ y $1 \leq num$ por lo que $\text{PRE}_S \equiv true$. Entonces comenzamos particionando $S_{VIS} = S_{IS} \mathrel{\widehat{=}} \mathbb{Z}$ de la siguiente forma:

$$S_1 \mathrel{\widehat{=}} \{x \in S_{VIS} \mid x < 1\}$$
$$S_2 \mathrel{\widehat{=}} \{x \in S_{VIS} \mid 1 \leq x\}$$

Derivando un ATC de cada clase de prueba nos aseguramos que vamos a testear P en sus dos comportamientos básicos: no ejecuta el ciclo (S_1) y ejecuta el ciclo (S_2). Pero ahora podemos suponer que el programador no tuvo en cuenta los desbordes de enteros y que por lo tanto el programa se podría comportar mal para valores de `num` mayores a $2^{31}-1$ pero bien para los menores. Es decir que S_2 no sería una clase de equivalencia. Entonces particionamos S_2 así:

$$S_{21} \mathrel{\widehat{=}} \{x \in S_2 \mid x \leq 2^{31}-1\}$$
$$S_{22} \mathrel{\widehat{=}} \{x \in S_2 \mid 2^{31}-1 < x\}$$

Claramente cualquier ATC derivado de S_{21} va a testear el ciclo que implementa P. Por otro lado sabemos que es común cometer errores en los casos bordes de los ciclos. Entonces P podría comportarse mal para $num = 1$ pero comportarse bien para $1 < num$; es decir, S_{21} no sería una clase de equivalencia. Por este motivo particionamos S_{21} de la siguiente forma obteniendo el árbol de testing de la Figura 4:

$$S_{211} \mathrel{\widehat{=}} \{x \in S_{21} \mid x = 1\}$$
$$S_{212} \mathrel{\widehat{=}} \{x \in S_{21} \mid 1 < x\}$$

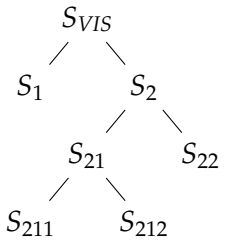

Figura 4: Árbol de testing para el programa de la variable `num`

Ahora no tenemos motivos razonables para suponer que las hojas del árbol no se comportarán como clases de equivalencia. . . ¿y si *num* es menor a $-(2^{31})$?

Los siguientes son posibles ATC del árbol de la Figura 4:

$S_1^{ATC} \widehat{=} num \mapsto 0$
$S_{211}^{ATC} \widehat{=} num \mapsto 1$
$S_{212}^{ATC} \widehat{=} num \mapsto 2$
$S_{22}^{ATC} \widehat{=} num \mapsto 2^{31}$

Automatización de Grounding. Como mencionamos más arriba, Grounding requiere resolver el mismo problema de satisfacibilidad que Podar. La única diferencia es que Grounding genera una solución para un predicado satisfacible mientras que Podar elimina la hoja del árbol si el predicado es insatisfacible. Como consecuencia de esto, la automatización de Grounding está sujeta *casi* a las mismas restricciones que la automatización de Podar. Es decir, la automatización de Grounding depende de la decidibilidad de la teoría sobre la cual se escribe S. Lo que diferencia Grounding de Podar es que si la teoría es *semi-decidible* entonces es posible automatizar Grounding, aunque no es posible automatizar Podar. Una teoría es semi-decidible si existe un algoritmo que en un tiempo finito es capaz de determinar si una fórmula cualquiera de la teoría es satisfacible cuando la fórmula lo es—pero puede no terminar si la fórmula es insatisfacible. Como a Grounding solo le importan las clases de prueba satisfacibles, la semi-decidibilidad de la teoría resulta suficiente—aunque es posible que queden hojas para las cuales no fue posible encontrar una solución lo que puede deberse a que la hoja es insatisfacible o a que el algoritmo necesita más tiempo para encontrar una solución.

2. Tácticas de testing disponibles en {*log*}-TTF

En esta sección vamos a explicar cuatro de las seis tácticas de testing que están implementadas en {*log*}-TTF[11]. La Tabla 1 resume la utilidad de cada táctica. Las dos primeras (DNF y SP) son útiles en la mayoría de las especificaciones, son muy generales y generan particiones clave para una buena cobertura. Suelen ser las primeras en ser aplicadas[12]. Las restantes tienen un

[11] Las dos restantes (ST y EX) las deben leer en el manual de usuario de {*log*}.
[12] En {*log*}-TTF es obligatorio aplicar primero DNF.

Tabla 1: Seis tácticas de testing implementadas en {log}-TTF

Sigla	Nombre	Foco
DNF	Formal normal disyuntiva	Estructura lógica de la especificación
SP	Particiones estándares	Operadores conjuntistas y relacionales
SC	Cardinalidad de conjuntos	Variables de tipo conjunto o relacional
II	Intervalos enteros	Variables de tipo entero
ST	Tipos suma	Variables de tipo suma disyunta
EX	Cuantificador existencial restringido	Cuantificador existencial restringido

Tabla 2: Comandos de usuario para aplicar tácticas en {log}-TTF. En los comandos: l es la etiqueta de un nodo del árbol; v es una variables de entrada que aparece en la cabecera de la operación. Todas las tácticas particionan todas las hojas del subárbol cuya raíz es l.

Táctica	Comando	Argumentos
DNF	`applydnf(`$p(i_1,\ldots,i_n)$`)`	p es la cabecera de una operación; i_k es un nombre de variable usado como argumento en la definición de p. i_k es una variable de entrada (no de estado) de p.
SP	`applysp(`l`,`$c(a_1,\ldots,a_n)$`)`	$c(a_1,\ldots,a_n)$ es un predicado atómico tal cual aparece en la operación; cada a_k debe estar escrito tal cual aparece en la operación, aun con los mismos nombres de variables.
SC	`applysc(`l`,`v`)`	v es una variable de tipo conjunto o relación.
II	`applyii(`l`,`v`,`s`)`	v es una variable de tipo entero; s es una lista no vacía de números enteros.
ST	`applyst(`l`,`v`)`	v es una variable de tipo suma.
EX	`applyex(`l`,`e`)`	$e = exists(x \in D, \phi)$ es un REC que aparece en la operación; x, D y ϕ deben escribirse igual que en la operación, aun con los mismos nombres de variables.

ámbito de aplicación más restringido y se las suele utilizar para afinar la cobertura.

La Tabla 2 muestra los comandos de usuario de {log}-TTF para aplicar cada táctica. En las tácticas donde es necesario escribir un término tal cual aparece en la especificación (e.g. SP y EX) se recomienda usar copiar-y-pegar.

Para resolver la parte de testing del TP aun deben leer la Sección 7 del apunte que explica cómo traducir de B a {log} y la Sección 13.6 del manual de usuario de {log}.

2.1. Ejemplo guía

Para ejemplificar algunas tácticas vamos a usar como ejemplo el forgrama {log} de la operación 'extraer' del sistema de cajas de ahorro que vimos en Ingeniería de Software 1 cuando introdujimos el método B. Para hacer el problema un poco más complejo agregamos el siguiente

```
parameters([MaxW,MinB]).

variables([Sa]).

withdrawOk(MaxW,MinB,Sa,N,A,Sa_,E) :-
  [N,B] in Sa &
  0 < A & A =< B &
  (MaxW < A implies MinB < B) &
  B1 is B - A &
  oplus(Sa,{[N,B1]},Sa_) &
  E = ok.

withdrawE1(Sa,N,Sa,E) :- comp({[N,N]},Sa,{}) & E = error1.

withdrawE2(Sa,A,Sa,E) :- neg(0 < A) & E = error2.

withdrawE3(MaxW,MinB,Sa,N,A,Sa,E) :-
  [N,B] in Sa &
  neg(A =< B & (MaxW < A implies MinB < B)) &
  E = error3.

operation(withdraw).
withdraw(MaxW,MinB,Sa,N,A,Sa_,E) :-
  withdrawOk(MaxW,MinB,Sa,N,A,Sa_,E)
  or withdrawE1(Sa,N,Sa,E)
  or withdrawE2(Sa,A,Sa,E)
  or withdrawE3(MaxW,MinB,Sa,N,A,Sa,E).
```

Figura 5: Especificación {log} de la operación `withdraw`

requerimiento:

- *No se permiten extracciones de más de $1.000 a menos que el saldo sea superior a $10.000.*

La especificación {*log*} se puede ver en la Figura 5. No declaramos invariantes ni el estado inicial ni damos el tipo de los predicados (aunque son obligatorios en una especificación {*log*}) porque no son relevantes para la aplicación de tácticas de testing. Los axiomas para establecer que `MaxW` y `MinB` valen 1000 y 10000 no los mostramos pero sí son tenidos en cuenta por {*log*}-TTF. Noten que en el predicado `withdrawOk` usamos una implicación lo que no es muy usual pero en este caso ayuda a explicar la primera táctica de testing. Por la misma razón usamos la negación (`neg`) en los predicados `withdrawE2` y `withdrawE3`.

Las siguientes son las designaciones asociadas a la especificación.

El saldo mínimo para hacer extracciones grandes $\approx$ *MinB*

Las extracciones grandes son mayores o iguales a $\approx$ *MaxW*

Una operación terminó con éxito $\approx$ *ok*

Una operación terminó con un error $\approx$ *error*1

Una operación terminó con un error $\approx$ *error*2

Una operación terminó con un error $\approx$ *error*3

La cuenta n tiene saldo $b \approx (n, b) \in Sa$

Un cliente extrae la suma A de la cuenta $N \approx withdraw(N, A)$

En `withdrawOk` usamos `[N,B] in Sa` para decir que `N` pertenece al dominio de `Sa` y que el saldo de esa cuenta es `B`. De esta forma no forzamos a {*log*} a calcular el dominio de `Sa` ni la imagen de `N` mediante `applyTo`. La precondición `(MaxW < A implies MinB < B)` formaliza el nuevo requerimiento enunciado más arriba. El predicado `withdrawE1` corresponde a la negación de la precondición `[N,B] in Sa`. Observen que no se puede usar `neg([N,B] in Sa)` porque `B` es una variable existencial. Dejamos como ejercicio comprender por qué `comp({[N,N]},Sa,{})` es la negación de `[N,B] in Sa`. En `withdrawE2` especificamos la negación de `0 < A`. Para poder calcular la negación de las precondiciones restantes primero necesitamos calcular el saldo de la cuenta `N` por lo que volvemos a usar `[N,B] in Sa` en `withdrawE3`.

Cuando sea posible vamos a usar `withdraw` como ejemplo para aplicar las tácticas de testing que introducimos a continuación.

El forgrama {*log*} completo lo pueden ver el Apéndice A.

2.2. Forma normal disyuntiva (DNF)

La táctica conocida como DNF consiste en escribir el predicado de la operación en forma normal disyuntiva. Es decir, se escribe el predicado como una disyunción de conjunciones de literales[13]. Luego se toma la precondición de cada término de la disyunción. Esas precondiciones son los predicados que caracterizan la partición. Los primeros en proponer DNF como una táctica de MBT fueron Jeremy Dick y Alain Faivre en 1993 [7]. Más formalmente, sea:

$$(q_{11} \wedge \cdots \wedge q_{1n_1}) \vee \cdots \vee (q_{k1} \wedge \cdots \wedge q_{kn_k})$$

la especificación de la operación en DNF donde todos los q_{ij} son literales. Entonces, los predicados que caracterizan la partición generada por la táctica DNF son los siguientes:

$$p_1^{\mathsf{DNF}} \mathrel{\widehat{=}} \textsc{pre}\ q_{11} \wedge \cdots \wedge q_{1n_1}$$

$$\vdots$$

$$p_k^{\mathsf{DNF}} \mathrel{\widehat{=}} \textsc{pre}\ q_{k1} \wedge \cdots \wedge q_{kn_k}$$

Por ejemplo, `withdraw` no está en DNF puesto que `withdrawOk` usa `implies` y `withdrawE3` tiene una negación de una conjunción dentro de la cual además está `implies`. Para no extender la presentación veamos la DNF solo de `withdrawOk`:

```
withdrawOk1(MaxW,MinB,Sa,N,A,Sa_,E) :-
  [N,B] in Sa &
  0 < A & A =< B &
```

[13] Un literal es un predicado atómico o la negación de un predicado atómico.

```
    A =< MaxW &
    B1 is B-A &
    oplus(Sa,{[N,B1]},Sa_) &
    E = ok.

withdrawOk2(MaxW,MinB,Sa,N,A,Sa_,E) :-
    [N,B] in Sa &
    0 < A & A =< B &
    MinB < B &
    B1 is B-A &
    oplus(Sa,{[N,B1]},Sa_) &
    E = ok.
```

Es decir, `withdrawOk1` considera el caso donde el monto a extraer (`A`) es menor o igual al monto estipulado como 'extracción grande' (`MaxW`) en cuyo caso la extracción se puede efectuar sin otra consideración. Por otro lado, `withdrawOk2` captura la situación donde el saldo de la cuenta (`B`) es mayor al monto estipulado como saldo mínimo para grandes extracciones (`MinB`) por lo cual la extracción se puede realizar. En otras palabras, `withdrawOk1` especifica una extracción 'chica' y `withdrawOk2` especifica una extracción de una cuenta con un saldo 'grande'.

El comando {*log*}-TTF para aplicar DNF a `withdraw` es el siguiente:

```
applydnf(withdraw(N,A))
```

puesto que `N` y `A` son las variables de entrada de la operación. Observen que *no* escribimos las variables de estado anterior, solo las variables de entrada propiamente dichas. `applydnf` calcula automáticamente la DNF de la operación, calcula las precondiciones y genera las clases de prueba correspondientes.

Este comando genera las seis clases de prueba[14] que se muestran en la Figura 6. Primero, notar que no se obtiene una partición del VIS, puesto que `withdraw_dnf_1` y `withdraw_dnf_2` se solapan—más abajo profundizaremos sobre esta cuestión. Segundo, notar que si tomamos un caso de prueba de cada una de las clases estaremos probando el sistema en todas las situaciones más importantes:

1. `withdraw_dnf_1`: se extrae una suma de dinero positiva pero no superior al saldo de la cuenta ni al límite de extracción.
2. `withdraw_dnf_2`: se extrae una suma de dinero positiva pero no superior al saldo de la cuenta en tanto esta tiene un saldo que supera el saldo mínimo para grandes extracciones.
3. `withdraw_dnf_3`: se intenta extraer de una cuenta no registrada en el banco.
4. `withdraw_dnf_4`: se quiere extraer un monto superior al saldo de la cuenta.
5. `withdraw_dnf_5`: el monto a extraer no es positivo.
6. `withdraw_dnf_6`: se quiere hacer una extracción grande sin tener un saldo suficientemente alto.

Como se puede apreciar con el ejemplo anterior, DNF es una táctica clave para obtener una cobertura funcional básica. En efecto, al llevar la especificación a DNF se obtiene un predicado

[14] El código {*log*} realmente generado incluye información que decidimos omitir para simplificar la presentación. Lo mismo vale para las clases de prueba que se muestran en las siguientes secciones.

```
withdraw_dnf_1(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A.

withdraw_dnf_2(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & MinB < B & B1 is B - A.

withdraw_dnf_3(MaxW,MinB,Sa,N,A) :- comp({[N,N]},Sa,{}).

withdraw_dnf_4(MaxW,MinB,Sa,N,A) :- A =< 0.

withdraw_dnf_5(MaxW,MinB,Sa,N,A) :- [N,B] in Sa &  B < A.

withdraw_dnf_6(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & MaxW < A & B =< MinB.
```

Figura 6: Clases de prueba generadas al aplicar DNF a `withdraw`

que es una suerte de sentencia **IF-THEN-ELSE** o **SELECT** de B. Luego, tomar la precondición de cada término resulta equivalente a considerar las condiciones de esas sentencias. Por lo tanto, cada clase de prueba que genera DNF corresponde a una de las condiciones que disparan o habilitan la parte **THEN** de las sentencias condicionales. En consecuencia, habrá casos de prueba que ejecutarán, al menos una vez, cada una de las postcondiciones según todas las formas posibles de dispararlas.

{*log*}-TTF conjuga todos los axiomas declarados en la especificación a todas las clases de prueba generadas por la aplicación de la táctica DNF—aunque en la Figura 6 no los incluimos para simplificar la presentación. De esta forma, cuando se generan casos de prueba los parámetros de la especificación que aparecen en cada clase de prueba toman valores consistentes con los axiomas.

La táctica DNF toma en cuenta la estructura lógica de la especificación pero no considera ni los operadores matemáticos que se usan en la especificación ni los tipos de las variables que intervienen. Por consiguiente deja fuera del análisis cuestiones importantes que son cubiertas por otras tácticas.

Como mencionamos más arriba DNF no siempre genera una partición porque en general no se puede asegurar la propiedad (4). Para que DNF genere una partición, durante la ejecución del algoritmo que reescribe la operación en DNF, se debe aplicar la siguiente reescritura luego de que se hayan eliminado todos los conectores lógicos excepto $\wedge$ y $\vee$ y la fórmula solo tenga literales:

$$p \vee q \rightarrow p \wedge q \vee p \wedge \neg q \vee \neg p \wedge q$$

Si se opera de esta forma, una vez que se han generado las clases de prueba se debe podar el árbol pues, en general, se generarán clases insatisfacibles. Esta poda no es necesaria si se usa el algoritmo estándar[15].

En {*log*}-TTF DNF es la primera táctica que se aplica.

[15] Estas y otras mejoras y extensiones a {*log*}-TTF pueden constituir una tesina.

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

Figura 7: Partición estándar para expresiones de la forma $R \lhd\!\!+ G$

2.3. Particiones estándares (SP)

Particiones estándares (SP) es una táctica de testing propuesta por Stocks en su tesis de doctorado [12]. Consiste en definir una *partición estándar* para el dominio de cada operador de la teoría de conjuntos y relaciones binarias.

La Figura 7 muestra la partición estándar definida por Stocks para el operador 'actualizar' del lenguaje B ($\lhd\!\!+$). Dado que este operador tiene tipo $Rel \times Rel \rightarrow Rel$ el dominio es $Rel \times Rel$ por lo que la partición indica condiciones a cumplir por las dos relaciones. Como puede apreciarse, Stocks propone ocho subconjuntos para la partición combinando diferentes condiciones sobre R y G. Al usar esta partición se generarán ocho clases de prueba aunque algunas de ellas pueden ser insatisfacibles en algunos casos.

La operación `withdraw` usa el operador `oplus`, que es el equivalente {*log*} a $\lhd\!\!+$, por lo que podemos aplicar SP. La idea al aplicar SP a uno de estos operadores es poder testear lo mejor posible la implementación de ese operador. Si ya tenemos las clases de prueba de la Figura 6, ¿cuál o cuáles de ellas deberíamos particionar con SP aplicada sobre `oplus(Sa,{[N,B1]},Sa_)`? La respuesta la tenemos que buscar en el criterio de aplicación de tácticas que enunciamos al final de la Sección 1.4.1. Es decir, tenemos que particionar las clases cuyos predicados impliquen la precondición que habilita la ejecución de `oplus(Sa,{[N,B1]},Sa_)` y que compartan alguna variable de entrada con las de ese predicado. La precondición que habilita la ejecución de ese `oplus` es la de `withdrawOk` la cual a su vez se manifiesta en dos clases de prueba luego de aplicar DNF: `withdraw_dnf_1` y `withdraw_dnf_2`. Ambas clases de prueba comparten variables de entrada con el `oplus` que queremos testear: `Sa` y `N`. El resto de las clases de prueba no satisfacen este criterio. Por lo tanto solo tiene sentido particionar con la SP de $\lhd\!\!+$ las clases `withdraw_dnf_1` y `withdraw_dnf_2`.

El comando {*log*}-TTF para aplicar esta táctica a la clase `withdraw_dnf_1` es el siguiente:

```
applysp(withdraw_dnf_1,oplus(Sa,{[N,B1]},Sa_))
```

Para particionar `withdraw_dnf_2` tenemos que ejecutar un comando similar. Si quisiéramos particionar todas las clases podríamos ejecutar un único comando:

```
applysp(withdraw_vis,oplus(Sa,{[N,B1]},Sa_))
```

porque en este caso {*log*}-TTF entiende que hay que particionar todas las hojas del (sub)árbol cuya raíz es `withdraw_vis`.

El primer comando `applysp` genera las ocho clases de prueba que se muestran en la Figura 8. Si miran con atención van a ver que la primera línea es la misma en todas las clases de prueba y es igual al predicado de la clase `withdraw_dnf_1`. Esto es, {*log*}-TTF expande automáticamente los predicados de los nodos intermedios del árbol de testing como hicimos en (8) en la Sección

1.4.3. Las líneas restantes de cada clase corresponden a cada una de las condiciones de la Figura 7 donde R se sustituye por `Sa` y G por `{[N,B1]}`. Por ejemplo, la primera condición de la SP de $\lhd\!\!+$ es $R = \emptyset, G = \emptyset$ que corresponde a la segunda línea de `withdraw_sp_11` luego de ejecutar la sustitución indicada: `Sa = {} & {[N,B1]} = {}`. También pueden ver que algunas clases (e.g `withdraw_sp_14`) hacen uso del predicado `let/3` para evitar la introducción de variables existenciales—revisar Sección 4.6 del apunte B a {*log*} y Sección 3.5 del manual de usuario.

Otra cuestión que es importante notar es que varias de las clases de prueba son insatisfacibles. Algunas lo son por la aplicación de SP a `oplus(Sa,{[N,B1]},Sa_)` y otras debido a la conjunción con el predicado de `withdraw_dnf_1`. Un ejemplo del primer motivo es `withdraw_sp_11` debido al predicado `{[N,B1]} = {}` que es trivialmente insatisfacible. Mientras que un ejemplo de la segunda es `withdraw_sp_12` puesto que `withdraw_dnf_1` pide `[N,B] in Sa` pero la segunda condición de SP para $\lhd\!\!+$ pide `Sa = {}`. En definitiva luego de aplicar SP también a `wihdraw_dnf_2` y efectuar la poda (comando `prunett`) se obtiene el árbol de testing mostrado en la Figura 9.

Como podemos ver, esta combinación de tácticas sugiere testear la implementación de `wihdraw` con ocho casos de prueba. Conceptualmente, ¿qué nuevas condiciones vamos a testear con las nuevas clases de prueba? Recordemos que `withdraw_dnf_1` testea la situación descrita en el ítem 1 de la Sección 2.2. Las hojas de `withdraw_dnf_1` testean esa situación de la siguiente forma:

1. `withdraw_sp_14`: se extrae una suma de dinero positiva pero no superior al saldo de la cuenta ni al límite de extracción, cuando la cuenta es la única cuenta del banco.

   La última parte corresponde al predicado:

   ```
   let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2)
   ```

   que es equivalente a $\mathrm{dom}(Sa) = \mathrm{dom}(\{N \mapsto B1\}) = \{N\}$.

2. `withdraw_sp_15`: se extrae una suma de dinero positiva pero no superior al saldo de la cuenta ni al límite de extracción, cuando hay más de una cuenta en el banco.

   La última parte corresponde al predicado:

   ```
   let([D1,D2],
       dom(Sa,D1) & dom({[N,B1]},D2), subset(D2,D1) & D1 neq D2)
   ```

   que es equivalente a $\mathrm{dom}(\{N \mapsto B1\}) = \{N\} \subset \mathrm{dom}(Sa)$.

Es decir que SP agrega condiciones de test significativas, una de las cuales testea el programa en un caso borde (`withdraw_sp_14`).

El motivo que llevó a Stocks a proponer SP es que cada operador matemático a nivel de la especificación esconde una implementación compleja a nivel del programa. Entonces, dividir la especificación únicamente teniendo en cuenta su estructura lógica (como lo hace DNF) no es suficiente para revelar errores en la implementación de esos operadores matemáticos. SP complementa DNF pues la primera actúa sobre la matemática de S mientras que la segunda lo hace sobre su estructura lógica. De todas formas, SP tiene en cuenta el sistema de tipos solo de manera superficial puesto que actúa solo sobre operadores conjuntistas y relacionales. Las siguientes tácticas apuntan a utilizar más en profundidad el sistema de tipos.

En este momento {*log*}-TTF implementa SP para nueve operadores: `un` ($\cup$), `inters` ($\cap$), `diff` ($\setminus$), `oplus` ($\lhd\!\!+$), `dres` ($\lhd$), `dares` ($\not\lhd$), `rres` ($\rhd$), `rares` ($\not\rhd$) y `rimg` (imagen relacional, $R[\_]$). Las particiones se pueden ver en el archivo `ttf_sp.pl` que forma parte de la distribución

```
withdraw_sp_11(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa = {} & {[N,B1]} = {}.

withdraw_sp_12(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa = {} & {[N,B1]} neq {}.

withdraw_sp_13(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]}={}.

withdraw_sp_14(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).

withdraw_sp_15(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2],
      dom(Sa,D1) & dom({[N,B1]},D2), subset(D2,D1) & D1 neq D2).

withdraw_sp_16(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), disj(D2,D1)).

withdraw_sp_17(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2],
      dom(Sa,D1) & dom({[N,B1]},D2), subset(D1,D2) & D1 neq D2).

withdraw_sp_18(MaxW,MinB,Sa,N,A) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2],
      dom(Sa,D1) & dom({[N,B1]},D2),
      ndisj(D1,D2) & nsubset(D2,D1) & nsubset(D1,D2)).
```

Figura 8: Clases de prueba resultantes al particionar `withdraw_dnf_1` aplicando SP sobre el predicado `oplus(Sa,[N,B1],Sa_)`

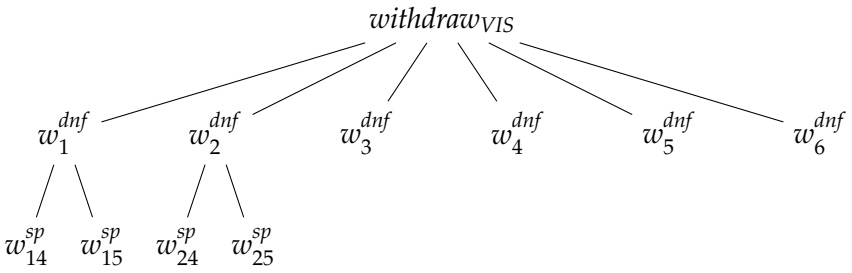

Figura 9: Árbol de testing de `withdraw` *luego de aplicar DNF, SP y podar*

de $\{log\}$. Es posible definir particiones para otros operadores, en algunos casos aplicando una técnica propuesta por Stocks llamada *propagación de subdominios* [12][16].

2.4. Cardinalidad de conjuntos (SC)

La táctica de testing denominada SC genera una partición asociando diferentes cardinalidades a una variable de tipo conjunto (lo que incluye a las relaciones binarias, funciones parciales, conjuntos ordenados y arreglos). Formalmente, sea A una variable de tipo conjunto entonces SC genera una partición caracterizada por los siguientes predicados:

$$p_1^{\mathsf{SC}} \mathrel{\widehat{=}} A = \emptyset \qquad \text{[es decir } |A| = 0\text{]}$$
$$p_2^{\mathsf{SC}} \mathrel{\widehat{=}} A = \{n\} \qquad \text{[donde } n \text{ es una variable nueva; es decir } |A| = 1\text{]}$$
$$p_3^{\mathsf{SC}} \mathrel{\widehat{=}} A = \{n_1, n_2 \mathbin{/} N\} \wedge n_1 \neq n_2 \qquad \text{[donde } n_i, N \text{ son variables nuevas; es decir } 1 < |A|\text{]}$$

Recordar que $\{x \mathbin{/} A\}$ es el constructor de conjuntos extensionales provisto por $\{log\}$ que equivale al término $\{x\} \cup A$—Sección 4.3.1 del apunte B a $\{log\}$. Entonces $\{n_1, n_2 \mathbin{/} N\}$ representa el conjunto $\{n_1, n_2\} \cup N$. Al pedir $n_1 \neq n_2$ se fuerza que $\{n_1, n_2 \mathbin{/} N\}$ tenga al menos dos elementos.

¿Por qué uno querría generar casos de prueba que incluyan conjuntos con cardinalidad 0, 1 y mayor a 1? Porque en general un conjunto se va a implementar con alguna estructura de datos que va a ser recorrida dentro de un ciclo o por medio de recursividad cada vez que haya que buscar, agregar, eliminar o modificar un elemento. En consecuencia es razonable testear el programa haciendo que el mecanismo de recorrido:

1. No ejecute ninguna iteración (caso cardinalidad cero)
2. Ejecute una única iteración (caso cardinalidad uno)
3. Ejecute más de una iteración (caso cardinalidad mayor a uno)

Se podrían exigir casos de prueba que hagan que el recorrido ejecute más iteraciones. Sin embargo, en la mayoría de los casos no parece haber un fundamento sólido para pedir más casos de prueba. Si el algoritmo que se quiere testear tiene un comportamiento diferente luego de la segunda iteración, es muy probable que eso se refleje de alguna forma en la especificación. Por ejemplo consideremos un programa que tiene que imprimir líneas de texto en una terminal de 80 columnas. ¿Funcionará bien el programa si le pedimos imprimir una línea con más de 80

[16] Esta es otra área donde $\{log\}$-TTF se puede mejorar y extender como parte de una tesina.

```
withdraw_sc_31(N,A,Sa,MaxW,MinB) :-
  comp({[N,N]},Sa,{}) &
  Sa = {}.

withdraw_sc_32(N,A,Sa,MaxW,MinB) :-
  comp({[N,N]},Sa,{}) &
  Sa = {X}.

withdraw_sc_33(N,A,Sa,MaxW,MinB) :-
  comp({[N,N]},Sa,{}) &
  Sa = {X,Y / R} & X neq Y.
```

Figura 10: Clases de prueba generadas al particionar `withdraw_dnf_3` aplicando SC sobre `Sa`

caracteres? En este caso uno podría pedir una partición diferente donde el número 80 sea usado para definirla.

En la especificación de `withdraw` tenemos la variable `Sa` que es la única de tipo conjunto. En particular tenemos la precondición `[N,B] in Sa` la cual seguramente se implementará con un recorrido sobre la estructura de datos que representará a `Sa` hasta dar con `N`, lo que permitirá recuperar `B`. Entonces para testear ese recorrido podríamos aplicar SC sobre `Sa`. ¿Tiene sentido aplicar SC sobre todas las hojas del árbol de la Figura 9? No, porque SP ya genera casos con `Sa` con dos o más elementos—e.g. `withdraw_sp_15`. Entonces tendríamos que buscar las hojas donde no sea evidente que `Sa` tomará valores con distintas cardinalidades. La búsqueda la tenemos que hacer sobre las hojas que no surgieron a raíz de aplicar SP y donde `Sa` sea una variable. La única clase que no cumple ese criterio es `withdraw_dnf_4`. El comando {*log*}-TTF para aplicar SC a `withdraw_dnf_3` sobre la variable `Sa` es el siguiente:

```
applysc(withdraw_dnf_3,Sa)
```

Las clases generadas se muestran en la Figura 10. Al igual que cuando aplicamos SP, la primera línea corresponde al predicado de la clase padre (`withdraw_dnf_3`) mientras que la segunda línea corresponde a los predicados generados por SC. La igualdad `Sa = {X}` puede dar lugar a confusión siendo que `Sa` es una función—i.e. un conjunto de pares ordenados. Sin embargo, entre el algoritmo de unificación de {*log*} y el sistema de tipos, `X` es unificada con un par ordenado de la forma `[X1,X2]` donde ambas son variables nuevas de tipos congruentes con el de `Sa`. En este caso no se generan clases insatisfacibles, aunque sí las hay al aplicar la misma táctica sobre `withdraw_dnf_5` y `withdraw_dnf_6`. ¿Cuáles son esas clases insatisfacibles?

El árbol de testing resultante al aplicar la misma táctica sobre las otras hojas se muestra en la Figura 11. Ahora el TTF nos está proponiendo testear la implementación de `withdraw` con doce casos de prueba. Nuevamente podemos poner en palabras las situaciones adicionales que vamos a testear. Tomemos como ejemplo `withdraw_dnf_3` que testea la situación descrita en en el ítem 3 de la Sección 2.2. Las hojas de esta clase testean la misma situación como sigue:

1. `withdraw_sc_31`: se intenta extraer de una cuenta no registrada en el banco, cuando el banco no tiene ninguna cuenta.

   La última parte corresponde al predicado `Sa = {}`.

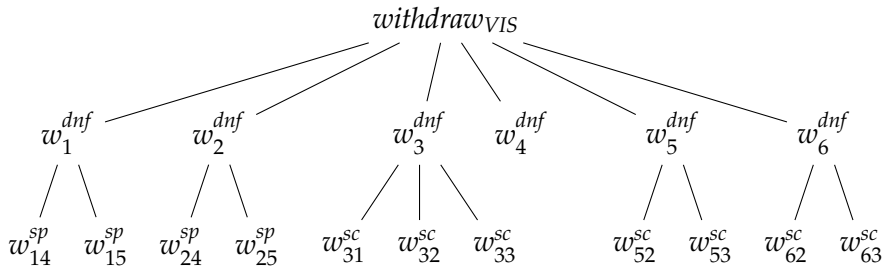

Figura 11: Árbol de testing de `withdraw` *luego de aplicar DNF, SP, SC y podar*

2. `withdraw_sc_32`: se intenta extraer de una cuenta no registrada en el banco, cuando el banco tiene una única cuenta.

   La última parte corresponde al predicado `Sa = {X}`.

3. `withdraw_sc_33`: se intenta extraer de una cuenta no registrada en el banco, cuando el banco tiene más de una cuenta.

   La última parte corresponde al predicado `Sa = {X,Y / R} & X neq Y`.

Es decir que nuevamente hemos agregado clases de test verdaderamente significativas, alguna de las cuales testean casos borde.

Actualmente {*log*}-TTF implementa SC de modo que siempre se genera la partición con tres clases de prueba que introdujimos al inicio de la sección. El usuario no puede modificar ese parámetro[17].

2.5. Intervalos de números enteros (II)

A nivel de especificación una variable numérica tiene tipo $\mathbb{Z}$ pero a nivel de implementación suele tener tipo **`short`**, **`int`**, **`long`**, etc. El problema es que $\mathbb{Z}$ es un conjunto infinito pero los tipos a nivel de implementación solo admiten un intervalo finito de números enteros. Rara vez los programadores usan enteros de longitud arbitraria que se asemejan mucho más a $\mathbb{Z}$. En consecuencia, los desarrolladores deberían incluir código para prevenir el desborde de enteros. La táctica de testing II tiene como aplicación principal generar casos de test que pongan a prueba ese código anti-desborde.

II espera del usuario una variable de tipo $\mathbb{Z}$, digamos v, y una lista ordenada de distintos números enteros, $[n_1, \ldots, n_k]$. De esta forma genera una partición cuyos predicados característicos son:

$$p_1^{\text{II}} \mathrel{\widehat{=}} v < n_1 \qquad p_2^{\text{II}} \mathrel{\widehat{=}} v = n_1$$
$$p_3^{\text{II}} \mathrel{\widehat{=}} n_1 < v < n_2 \qquad p_4^{\text{II}} \mathrel{\widehat{=}} v = n_2 \qquad p_5^{\text{II}} \mathrel{\widehat{=}} n_2 < v < n_3$$
$$\vdots$$
$$p_{2k-3}^{\text{II}} \mathrel{\widehat{=}} n_{k-2} < v < n_{k-1} \qquad p_{2k-2}^{\text{II}} \mathrel{\widehat{=}} v = n_{k-1} \qquad p_{2k-1}^{\text{II}} \mathrel{\widehat{=}} n_{k-1} < v < n_k$$
$$p_{2k}^{\text{II}} \mathrel{\widehat{=}} v = n_k \qquad p_{2k+1}^{\text{II}} \mathrel{\widehat{=}} n_k < v$$

[17] Es otra mejora que se puede introducir en {*log*}-TTF.

En particular para el caso más común de una lista de dos elementos, $[m,n]$, la partición es la siguiente:

$$p_1^{\text{II}} \mathrel{\widehat{=}} v < m \qquad p_2^{\text{II}} \mathrel{\widehat{=}} v = m \qquad p_3^{\text{II}} \mathrel{\widehat{=}} m < v < n \qquad p_4^{\text{II}} \mathrel{\widehat{=}} v = n \qquad p_5^{\text{II}} \mathrel{\widehat{=}} n < v$$

En `withdraw` tenemos la variable entera `A` que corresponde al monto que se quiere retirar. Si suponemos que esa variable se va a implementar con un `int` que toma valores en el intervalo $[-(2^{31}), 2^{31}-1]$ entonces podemos aplicar II. Nuevamente tenemos que decidir en qué parte del árbol de la Figura 11 vamos a aplicar la táctica. En este caso el objetivo es testear el código que realiza comparaciones con la variable `A` u operaciones matemáticas como suma o resta en las que intervenga esa variable. La única clase que no satisface esas condiciones es `withdraw_dnf_3` por lo que aplicaremos la táctica en los subárboles con raíz `withdraw_dnf_`$\langle i \rangle$ con i distinto de 3. El comando para aplicar II a, por ejemplo, el subárbol `withdraw_dnf_1` es:

```
applyii(withdraw_dnf_1,A,[-2147483648,2147483647])
```

De esta forma particionamos las clases `withdraw_sp_14` y `withdraw_sp_15` con un solo comando. Las clases que se obtienen al particionar `withdraw_sp_14` se pueden ver en la Figura 12. La primera línea de código de cada clase corresponde al predicado de `withdraw_dnf_1`; las dos siguientes al predicado de `withdraw_sp_14`; y la última al predicado de II. Está claro que las dos primeras clases son insatisfacibles debido a que ambas piden `0 < A`.

Luego de aplicar II a todos los subárboles que mencionamos antes y podar, obtenemos el árbol de testing de la Figura 13. De esta forma el TTF establece testear `withdraw` con 34 casos de prueba. Como hicimos en los casos anteriores vamos a dar una descripción coloquial de las nuevas situaciones que vamos a testear. Consideremos como ejemplo `withdraw_sp_15` que está descrita coloquialmente en el ítem 2 de la Sección 2.3. Las hojas de esa clase testean la misma situación como se indica a continuación (se indica en *itálica* la condición que se agrega):

1. `withdraw_ii_153`: se extrae una suma de dinero positiva pero no superior al saldo de la cuenta ni al límite de extracción *y estrictamente dentro del rango de* `int`, cuando hay más de una cuenta en el banco.
2. `withdraw_ii_154`: se extrae una suma de dinero positiva *igual al máximo* `int` pero no superior al saldo de la cuenta ni al límite de extracción, cuando hay más de una cuenta en el banco.
3. `withdraw_ii_155`: se extrae una suma de dinero positiva *que supera al máximo* `int` pero no superior al saldo de la cuenta ni al límite de extracción, cuando hay más de una cuenta en el banco.

Al igual que en los casos anteriores el TTF genera clases de prueba significativas que pueden detectar errores sutiles.

Como las dos tácticas de testing ST y EX no se pueden aplicar a `withdraw` podemos considerar que el de la Figura 13 es el árbol de testing definitivo. En consecuencia el paso siguiente debería ser generar los casos de test ejecutando el comando `gentc` de {*log*}-TTF. Los comandos usados para generar el árbol de la Figura 13 y los casos de prueba generados a partir de él, se muestran en el Apéndice B.

El comando `applyii` ordena la lista que le pasa el usuario así que no es necesario dar una lista ordenada. Por otra parte, el comando se podría mejorar de varias formas: debería ser posible pasar una expresión y no solo una variable; podría aceptar palabras como `int` o `long` o `c_long` que representan intervalos estándar que el comando aplica automáticamente.

```
withdraw_ii_141(N,A,Sa,MaxW,MinB) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).
  A < -2147483648.

withdraw_ii_142(N,A,Sa,MaxW,MinB) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).
  A = -2147483648.

withdraw_ii_143(N,A,Sa,MaxW,MinB) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).
  -2147483648 < A & A < 2147483647.

withdraw_ii_144(N,A,Sa,MaxW,MinB) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).
  A = 2147483647.

withdraw_ii_145(N,A,Sa,MaxW,MinB) :-
  [N,B] in Sa & 0 < A & A =< B & A =< MaxW & B1 is B - A &
  Sa neq {} & {[N,B1]} neq {} &
  let([D1,D2], dom(Sa,D1) & dom({[N,B1]},D2), D1 = D2).
  2147483647 < A.
```

Figura 12: Clases de prueba generadas al particionar `withdraw_sp_14` *aplicando II sobre* `A` *con la lista* [−2147483648, 2147483647]

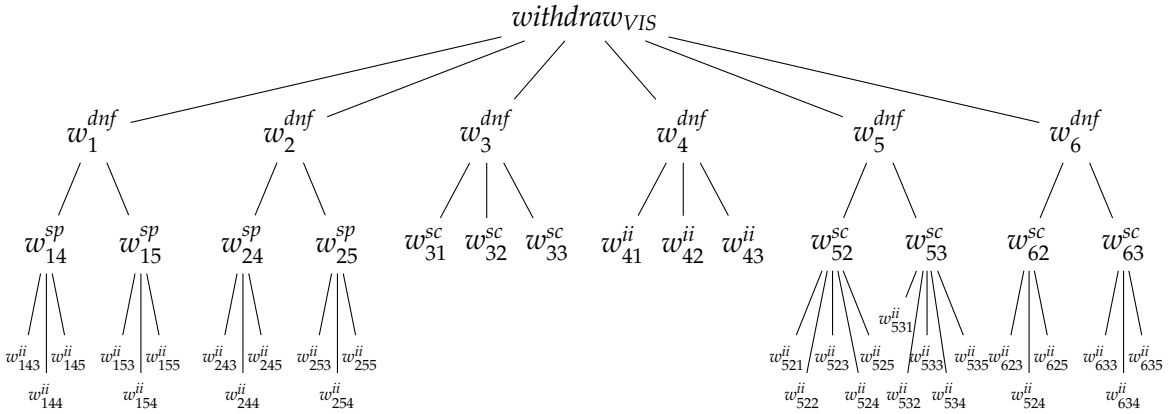

Figura 13: Árbol de testing de `withdraw` *luego de aplicar DNF, SP, SC, II y podar*

3. Otras tácticas de testing

Varias tácticas más se podrían implementar en {*log*}-TTF[18]. Los siguientes son algunos ejemplos:

- Mutación de especificaciones (SM). Es una táctica propuesta por Stocks en su tesis doctoral inspirada en el *testing por mutación*[19]. La idea consiste en suponer que el programador no implementó la especificación de una operación sino una versión ligeramente diferente, que Stocks llama *especificación mutante*. Por ejemplo, si la especificación dice $R \lhd\!\!+ \{x \mapsto y\}$ el programador se equivocó e implementó $R \cup \{x \mapsto y\}$. Entonces la táctica genera casos de prueba que distinguen la especificación original de la especificación mutante. De esta forma, si el programador cometió ese error es imposible que el caso generado por esta táctica no produzca una falla.
- Causa-efecto (CE). Esta táctica representa una de las formas más tradicionales de MBT. CE propone particionar el VIS partiendo de las postcondiciones (i.e. los efectos). Por ejemplo en `withdraw` uno podría particionar el VIS de forma que se alcancen todos los valores posibles de la variable de salida `E`.
- Relaciones y funciones (RF). La táctica RF propone particionar en base a considerar que una relación binaria usada en la especificación no es función, es función constante, es función inyectiva, es función pero no es constante ni inyectiva. La idea es que tal vez el programador no previó la no repetición de elementos en el dominio o la repetición o la unicidad de elementos en el rango.
- Conjuntos de números enteros (IS). Esta táctica propone particionar un conjunto de enteros usado en la especificación con un conjunto de números negativos, el conjunto unitario con el cero, un conjunto de números positivos, un conjunto que incluye números negativos, el cero y números positivos.
- Distinguir operadores aritméticos (AO). Un error de programación posible es, por ejemplo, que en lugar de escribir $x + y$ el programador escribe $x \cdot y$. Si el programa se testea solo

[18] Esta es otra área donde {*log*}-TTF se puede mejorar y extender como parte de una tesina.
[19] 'mutation testing', en inglés.

con $x = y = 0$ no se va a producir una falla. Esta táctica asegura que se van a generar casos donde $x + y \neq x \cdot y$. AO es una forma especializada de SM.

Referencias

[1] Maximiliano Cristiá, Pablo Albertengo, Claudia S. Frydman, Brian Plüss, and Pablo Rodríguez Monetti. Tool support for the test template framework. *Softw. Test. Verification Reliab.*, 24(1):3–37, 2014.

[2] Maximiliano Cristiá, Pablo Albertengo, and Pablo Rodríguez Monetti. Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In José Luis Fiadeiro and Stefania Gnesi, editors, *SEFM*, pages 268–277. IEEE Computer Society, 2010.

[3] Maximiliano Cristiá, Alfredo Capozucca, and Gianfranco Rossi. {*log*}: From a constraint logic programming language to a formal verification tool. *Theory and Practice of Logic Programming*, pages 1–32, 2026.

[4] Maximiliano Cristiá, Diego Hollmann, Pablo Albertengo, Claudia S. Frydman, and Pablo Rodríguez Monetti. A language for test case refinement in the Test Template Framework. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 601–616. Springer, 2011.

[5] 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.

[6] Maximiliano Cristiá and Gianfranco Rossi. Integrating cardinality constraints into constraint logic programming with sets. *Theory Pract. Log. Program.*, 23(2):468–502, 2023.

[7] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Jim Woodcock and Peter Gorm Larsen, editors, *FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark, April 19-23, 1993, Proceedings*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 1993.

[8] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.

[9] Vahid Garousi, Alper Bugra Keles, Yunus Balaman, Zeynep Özdemir Güler, and Andrea Arcuri. Model-based testing in practice: An experience report from the web applications domain. *J. Syst. Softw.*, 180:111032, 2021.

[10] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul J. Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, 2009.

[11] Florian Sommer, Reiner Kriesten, and Frank Kargl. Survey of model-based security testing approaches in the automotive domain. *IEEE Access*, 11:55474–55514, 2023.

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

[13] Phil Stocks and David A. Carrington. A framework for specification-based testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.

[14] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[15] Jeremy Vanhecke, Xavier Devroey, and Gilles Perrouin. Abscon: A test concretizer for model-based testing. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, pages 15–22. IEEE, 2019.

A. Especificación completa de la operación `withdraw`

La siguiente es la especificación {*log*} completa para la operación `withdraw`. Tengan en cuenta que {*log*}-TTF incluye de forma automática los axiomas como precondiciones de las operaciones. Sin embargo, las invariantes no son tenidas en cuenta durante el testing. Por otro lado como {*log*} exige que una especificación tenga al menos un invariante entonces declaramos uno; sin dudas se puede declarar alguna otra invariante.

```
parameters([MaxW,MinB]).

variables([Sa]).

def_type(error,enum([ok,error1,error2,error3])).

axiom(vMaxW).
dec_p_type(vMaxW(int)).
vMaxW(MaxW) :- MaxW = 1000.

axiom(vMinB).
dec_p_type(vMinB(int)).
vMinB(MinB) :- MinB = 10000.

invariant(saPfun).
dec_p_type(saPfun(rel(aid,int))).
saPfun(Sa) :- pfun(Sa).

initial(init).
dec_p_type(init(rel(aid,int))).
init(Sa) :- Sa = {}.

dec_p_type(
  withdrawOk(int,int,rel(aid,int),aid,int,rel(aid,int),error)
).
withdrawOk(MaxW,MinB,Sa,N,A,Sa_,E) :-
  [N,B] in Sa & dec([B,B1],int) &
  0 < A & A =< B &
  (MaxW < A implies MinB < B) &
  B1 is B - A &
  oplus(Sa,{[N,B1]},Sa_) &
  E = ok.

dec_p_type(withdrawE1(rel(aid,int),aid,rel(aid,int),error)).
withdrawE1(Sa,N,Sa,E) :- comp({[N,N]},Sa,{}) & E = error1.

dec_p_type(withdrawE2(rel(aid,int),int,rel(aid,int),error)).
withdrawE2(Sa,A,Sa,E) :- neg(0 < A) & E = error2.
```

```
dec_p_type(
  withdrawE3(int,int,rel(aid,int),aid,int,rel(aid,int),error)
).
withdrawE3(MaxW,MinB,Sa,N,A,Sa,E) :-
  [N,B] in Sa & dec(B,int) &
  neg(A =< B & (MaxW < A implies MinB < B)) &
  E = error3.

operation(withdraw).
dec_p_type(
  withdraw(int,int,rel(aid,int),aid,int,rel(aid,int),error)
).
withdraw(MaxW,MinB,Sa,N,A,Sa_,E) :-
  withdrawOk(MaxW,MinB,Sa,N,A,Sa_,E)
  or withdrawE1(Sa,N,Sa,E)
  or withdrawE2(Sa,A,Sa,E)
  or withdrawE3(MaxW,MinB,Sa,N,A,Sa,E).
```

B. Comandos y casos de prueba de `withdraw`

Se usaron los siguientes comandos para generar el árbol de testing de la Figura 13.

```
{log}=> applydnf(withdraw(N,A)).
{log}=> applysp(withdraw_dnf_1,oplus(Sa,{[N,B1]},Sa_)).
{log}=> prunett.
{log}=> applysp(withdraw_dnf_2,oplus(Sa,{[N,B1]},Sa_)).
{log}=> prunett.
{log}=> applysc(withdraw_dnf_3,Sa).
{log}=> prunett.
{log}=> applysc(withdraw_dnf_5,Sa).
{log}=> prunett.
{log}=> applysc(withdraw_dnf_6,Sa).
{log}=> prunett.
{log}=> applyii(withdraw_dnf_1,A,[-2147483648,2147483647]).
{log}=> prunett.
{log}=> applyii(withdraw_dnf_2,A,[-2147483648,2147483647]).
{log}=> prunett.
{log}=> applyii(withdraw_dnf_4,A,[-2147483648,2147483647]).
{log}=> prunett.
{log}=> applyii(withdraw_dnf_5,A,[-2147483648,2147483647]).
{log}=> prunett.
{log}=> applyii(withdraw_dnf_6,A,[-2147483648,2147483647]).
{log}=> prunett.
{log}=> gentc.
```

Los siguientes son los casos de prueba generados a partir del *script* anterior. Observen que a pesar de que DNF no genera una partición de `withdraw` ningún caso de prueba se repite.

```
withdraw_tc_143(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 1000 & N = an:n0 & Sa = {[an:n0,1000]}.

withdraw_tc_153(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 1000 & N = an:n0 &
 Sa = {[an:n0,1000],[an:n1,2]}.

withdraw_tc_243(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 10001 & N = an:n0 & Sa = {[an:n0,10001]}.

withdraw_tc_244(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n0 &
 Sa = {[an:n0,2147483647]}.

withdraw_tc_245(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n0 &
 Sa = {[an:n0,2147483648]}.

withdraw_tc_253(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 10001 & N = an:n0 &
 Sa = {[an:n0,10001],[an:n1,2]}.

withdraw_tc_254(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n0 &
 Sa = {[an:n0,2147483647],[an:n1,2]}.

withdraw_tc_255(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n0 &
 Sa = {[an:n0,2147483648],[an:n1,2]}.

withdraw_tc_31(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & N = an:n0 & Sa = {}.

withdraw_tc_32(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & N = an:n0 & Sa = {[an:n1,2]}.

withdraw_tc_33(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & N = an:n0 & Sa = {[an:n1,3],[an:n2,4]}.

withdraw_tc_41(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483649.

withdraw_tc_42(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483648.

withdraw_tc_43(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 0.

withdraw_tc_521(N,A,Sa,MaxW,MinB) :-
```

```
 MaxW = 1000 & MinB = 10000 & A = -2147483649 & N = an:n0 &
 Sa = {[an:n0,-2147483650]}.

withdraw_tc_522(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483648 & N = an:n0 &
 Sa = {[an:n0,-2147483649]}.

withdraw_tc_523(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483647 & N = an:n0 &
 Sa = {[an:n0,-2147483648]}.

withdraw_tc_524(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n0 &
 Sa = {[an:n0,2147483646]}.

withdraw_tc_525(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n0 &
 Sa = {[an:n0,2147483647]}.

withdraw_tc_531(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483649 & N = an:n1 &
 Sa = {[an:n1,-2147483650],[an:n0,0]}.

withdraw_tc_532(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483648 & N = an:n1 &
 Sa = {[an:n1,-2147483649],[an:n0,0]}.

withdraw_tc_533(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = -2147483647 & N = an:n1 &
 Sa = {[an:n1,-2147483648],[an:n0,0]}.

withdraw_tc_534(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n1 &
 Sa = {[an:n1,2147483646],[an:n0,0]}.

withdraw_tc_535(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n1 &
 Sa = {[an:n1,2147483647],[an:n0,0]}.

withdraw_tc_623(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 1001 & N = an:n0 & Sa = {[an:n0,0]}.

withdraw_tc_624(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n0 &
 Sa = {[an:n0,10000]}.

withdraw_tc_625(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n0 & Sa = {[an:n0,0]}.

withdraw_tc_633(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 1001 & N = an:n1 &
 Sa = {[an:n1,0],[an:n0,0]}.
```

```
withdraw_tc_634(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483647 & N = an:n1 &
 Sa = {[an:n1,10000],[an:n0,0]}.

withdraw_tc_635(N,A,Sa,MaxW,MinB) :-
 MaxW = 1000 & MinB = 10000 & A = 2147483648 & N = an:n1 &
 Sa = {[an:n1,0],[an:n0,0]}.
```