Apunte de clase

Introducción al Método B

Maximiliano Cristiá

Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Rosario – Argentina

© Maximiliano Cristiá – 2025 – Todos los derechos reservados

Índice

1. Cuestiones preliminares **3**

1.1. Tipos de datos en B 4

1.2. Dominio y aplicación de función 6

2. Cajas de ahorro – Requerimientos del usuario **6**

3. Cajas de ahorros – Especificación B **7**

3.1. Variables de estado 7

3.1.1. ¿Por qué no usar una lista en lugar de una función para *sa*? 10

3.2. Especificación de las operaciones sobre las cajas de ahorros 11

3.2.1. Abrir una caja de ahorros (transición de estados) 12

3.2.2. Depositar dinero en una caja de ahorros (transición de estados) 16

3.2.3. Retinar dinero de una caja de ahorros (transición de estados) 21

3.2.4. Consultar el saldo disponible (consulta de estado) 24

3.2.5. Cerrar una caja de ahorros (transición de estados) 24

4. Parámetros de configuración y sus restricciones **26**

5. Introducción a la verificación de máquinas de estado **28**

6. Tipos y conjuntos **33**

7. Designaciones **36**

8. Cada titular puede tener más de una caja de ahorros y viceversa **37**

8.1. Tipos elementales y variables de estado 38

8.2. Las operaciones 40

8.3. Transferencia de dinero entre dos cuentas – Composición secuencial 40

9. El sistema elije el número de cuenta — La sentencia ANY **42**

10. Ejercicios **44**

Más abajo vamos a introducir las cuestiones más importantes del método B por medio de un ejemplo donde formalizamos un conjunto de requerimientos como una especificación B. Sin embargo antes vamos a presentar algunas cuestiones preliminares sobre el método B.

Luego de estudiar este apunte deben leer los capítulos 1, 2, 4–7 y 9 del libro "The B-Method: an Introduction" de Steve Schneider disponible gratuitamente en la Universidad de Surrey. Pueden obviar todo lo relacionado a "weakest precondition".

1. Cuestiones preliminares

Las especificaciones B toman la forma de máquinas de estados descritas por medio de lógica de predicados y teoría de conjuntos. Una máquina de estados B consiste de:

1. *Un conjunto de estados*. El conjunto de estados se da al declarar *variables de estado*. Cada variable de estado tiene un nombre y un tipo. El conjunto de estados queda definido por todos los valores de todas las variables de estado

   Por ejemplo, si tenemos dos variables de estado, x de tipo $\mathbb{Z}$ y y de tipo $\{ok, no\}$, entonces algunos de los posibles estados de esta máquina de estados se pueden escribir usando notación matemática de la siguiente forma (esto NO es notación B):

   $$(x = 1, y = ok), (x = 4, y = ok), (x = 4, y = no), (x = 7, y = ok), \ldots$$

   Estos estados se podrían dibujar usando la notación gráfica clásica para máquinas de estado:

   Dos estados de una máquina de estados son iguales si cada variable toma el mismo valor en ambos estados. De esta forma, cada vez que una variable de estado cambia de valor se produce un cambio de estado.

2. *Un conjunto de operaciones de estado*. Las operaciones de estado se describen mediante lógica de predicados y teoría de conjuntos. Hay dos clases de operaciones de estado: *transiciones de estado* y *consultas de estado*. Las transiciones de estado hacen que una máquina de estados pase de un estado a otro, posiblemente tomando alguna entrada y produciendo alguna salida. Las consultas de estado no cambian el estado actual de la máquina mientras que producen alguna salida, posiblemente tomando algunas entradas.

   Si x es una variable de estado y una transición de estado cambia el valor de x, escribimos $x := expr$ queriendo decir que el nuevo valor de x es el que se obtiene al evaluar la expresión $expr$.

   Por ejemplo, podemos decir que la transición de estados T incrementa el valor de la variable de estados x en 3 mientras que no modifica la variable y. Entonces podemos escribir T usando matemática de la siguiente forma (esto NO es notación B):

   $$T \mathrel{\widehat{=}} [x := x + 3]$$

   Gráficamente, podemos decir que T es la transición que conecta los dos primeros estados mostrados más arriba:

En cualquier caso, T establece que la máquina de estos va a ir de un estado a otro donde el valor de x es tres unidades más grande que el valor anterior de x mientras que el valor de y permanece igual.

Cuando T transiciona del estado s_1 al estado s_2 decimos que s_1 es el *estado anterior* o el *estado de partida* y s_2 es el *estado siguiente* o el *estado de llegada*.

1.1. Tipos de datos en B

Como mencionamos más arriba, en B cada variable tiene un tipo. Los tipos en B son más o menos como los tipos de los lenguajes de programación aunque son más similares a los tipos usados en matemática. Si T es un tipo de B y x es una variable, entonces $x \in T$ significa que x es de o tiene tipo T.

En B tenemos los siguientes tipos:

- *Números*. En B existe el tipo de los números enteros, simbolizado con $\mathbb{Z}$, y el tipo de los números naturales, simbolizado con $\mathbb{N}$.

  Están disponibles los operadores aritméticos habituales: $+$, $-$, $*$, div, mod, $>$, $<$, $\leq$ y $\geq$.

- *Tipos enumerados o conjuntos enumerados*. Un tipo o conjunto enumerado es simplemente una enumeración de constantes. Un tipo enumerado se introduce de la siguiente forma:

  $$TYPE_NAME = \{const_1, \dots, const_n\}$$

  Por ejemplo, si estamos modelando el ciclo de trabajo de un cierto expediente podríamos necesitar el siguiente tipo enumerado:

  $$STATUS = \{requested, verified, aproved\}$$

  Esto significa que el expediente puede haber sido solicitado (*requested*), eventualmente será verificado (*verified*) y eventualmente será aprobado (*aproved*).

- *Tipos básicos o conjuntos básicos*. Un tipo básico es un conjunto de valores cuya forma o estructura es desconocida. Los tipos básicos se introducen simplemente escribiendo el nombre del tipo. Por ejemplo, si queremos trabajar con domicilios usamos el nombre *ADDR* para indicar el conjunto de todos los domicilios posibles. De esta forma, no nos importa si los domicilios son cadenas de caracteres o números o cualquier otra cosa.

- *Productos cartesianos*. Un tipo producto cartesiano es el producto cartesiano de dos tipos. Se declara como se muestra a continuación:

  $$T \times U$$

  donde T y U son dos tipos B. Un elemento de $T \times U$ es un par ordenado (x, y) donde x es de tipo T e y es de tipo U. Es decir, $T \times U$ es el conjunto de todos los pares ordenados (x, y) donde x es un elemento de T e y es un elemento de U.

Los productos cartesianos son útiles para agrupar características relacionadas de una cierta entidad. Por ejemplo, podemos modelar los datos personales de individuos como la combinación de un domicilio y una fecha de nacimiento:

$$ADDR \times DATE$$

Entonces si tenemos $p \in ADDR \times DATE$ significa que p es un par ordenado cuya primera componente es de tipo $ADDR$ y su segunda componentes es de tipo $DATE$.

- *Tipo conjunto potencia*. Un tipo conjunto potencia (o simplemente un tipo conjunto) es el tipo de todos los subconjuntos de un cierto tipo. Se denota con $\mathbb{P}T$, donde T es un tipo B. Los elementos de un tipo conjunto son conjuntos. Por ejemplo, si tenemos:

  $$\mathbb{P}\,STATUS$$

  entonces algunos elementos de este tipo son: $\{\}$, $\{requested\}$, $\{verified, aproved\}$, ...

  Los tipos conjunto se usan en casi todas las especificaciones B. Por ejemplo, la especificación B correspondiente a un programa que retorna el máximo de una lista de números se describe en realidad con un conjunto de números, y no con una lista de números. De hecho el programa retornará el mismo resultado sin importar el orden de los números ni si hay o no repeticiones en la lista. Por caso, el máximo de estas dos listas es el mismo:

  $$[3,5,1,3]$$
  $$[1,5,3,3]$$

  debido a que ambas listas contienen el mismo conjunto de números. Esto es así porque, en esencia, estas dos listas pueden representarse con el mismo conjunto: $\{5,1,3\}$. Entonces, si s es el conjunto que representa la lista de números de la cual queremos computar el máximo, la declaración de s es $s \in \mathbb{P}\mathbb{Z}$. Notar que si la declaración fuese $s \in \mathbb{Z}$, entonces s representa un número, no un conjunto de ellos.

  B provee una amplia gama de operadores de conjuntos que, cuando se los usa de forma inteligente, dan lugar a especificaciones claras, concisas y simples.

  Los conjuntos y sus operadores forman el corazón del lenguaje de especificación B.

- *Relaciones binarias*. $X \leftrightarrow Y$ denota el tipo de todas las relaciones binarias entre elementos de los tipos X e Y. Este tipo está definido como $X \leftrightarrow Y \mathrel{\widehat=} \mathbb{P}(X \times Y)$. Recordar que una relación binaria entre los conjuntos X e Y es cualquier subconjunto de $X \times Y$. En otras palabras, una relación binaria es un conjunto de pares ordenados.

  En consecuencia, si tenemos $R \in X \leftrightarrow Y$ entonces sabemos que R es un conjunto de pares ordenados tales que cada primera componente es un elemento de X y cada segunda componente es un elemento de Y. Esto es, R es un conjunto de la forma $\{(x_1,y_1),(x_2,y_2),\dots\}$.

- *Funciones parciales*. $X \nrightarrow Y$ denota el *conjunto* de todas las funciones parciales de X en Y. Una función parcial es una función que puede no estar definida en todo su dominio. En B las funciones parciales son conjuntos de pares ordenados. De hecho, tenemos lo siguiente:

  $$X \nrightarrow Y \subset X \leftrightarrow Y$$

  Es decir, toda función parcial es una relación binaria (aunque no toda relación binaria es una función parcial). Más precisamente:

  $$X \nrightarrow Y \mathrel{\widehat=} \{f \mid f \in X \leftrightarrow Y \land \forall x, y_1, y_2.((x,y_1) \in f \land (x,y_2) \in f \Rightarrow y_1 = y_2)\}$$

- *Secuencias o listas*. $\text{seq}\, X$ denota el *conjunto* de todas las secuencias (listas) de elementos de tipo X. En B las secuencias son conjuntos de pares ordenados. De hecho, tenemos lo siguiente:

  $$\text{seq}\, X \subset \mathbb{N} \nrightarrow X$$

  Más precisamente cualquier elemento de tipo $\text{seq}\, X$ es una función parcial cuyo dominio es el intervalo entero $1 \,..\, n$ para algún n (en B, $1 \,..\, n$ corresponde a $[1, n]$); n es la longitud de la secuencia. Formalmente:

  $$\text{seq}(X) \mathrel{\widehat{=}} \{s \mid s \in \mathbb{N} \nrightarrow X \land \exists n.(n \in \mathbb{N} \land \text{dom}(s) = 1 \,..\, n)\}$$

  A pesar de que en B las secuencias son conjuntos, el lenguaje proveer simplificaciones sintácticas que ayudan a trabajar con ellas:
  - La lista vacía se nota [ ].
  - Las listas explícitas se puede escribir entre corchetes: $[a_1, \ldots, a_n]$.

1.2. Dominio y aplicación de función

Aquí recordaremos un par de conceptos que deberían saber.

El *dominio* de una relación binaria es el conjunto de todas las primeras componentes de los pares ordenados que pertenecen a la relación. Formalmente, si $R \in X \leftrightarrow Y$ entonces:

$$\text{dom}(R) = \{x \mid x \in X \land \exists y.(y \in Y \land (x, y) \in R)\}$$

Si f es una función, entonces $f(x)$ denota la *aplicación* de f a x. Recordar que si $f \in X \nrightarrow Y$, entonces $f(x)$ podría no estar definida para toda $x \in X$.

2. Cajas de ahorro – Requerimientos del usuario

Los requerimientos del usuario del ejemplo que vamos a usar para introducir el método B consisten de las operaciones básicas sobre las cajas de ahorro de un banco. Esto es una simplificación de un sistema real que nos sirve como un ejemplo introductorio para estudiar la formalización de requerimientos.

- Cada cliente del banco se identifica con su documento de identidad.
- Cada cliente del banco puede tener a lo sumo una caja de ahorros.
- Cada caja de ahorros se identifica con la identidad de su titular.
- Cada caja de ahorros debe mantener el saldo actual.
- Cuando se abre una caja de ahorros su saldo inicial es cero.
- Una vez que se abrió una caja de ahorros es posible depositar una cantidad positiva de dinero; es posible extraer una cantidad positiva de dinero no mayor al saldo actual; y es posible consultar el saldo actual.
- Una caja de ahorros se puede cerrar solo si su saldo es cero.
- El saldo de una caja de ahorros nunca puede ser negativo.

3. Cajas de ahorros – Especificación B

Ahora vamos a dar la especificación B correspondiente a los requerimientos enunciados en la sección anterior. Vamos a proceder de la siguiente forma:

1. Vamos a definir las variables de estado
2. Vamos a definir el conjunto de estados de la máquina de estados
3. Vamos a especificar las operaciones de la máquina de estados

La máquina de estados se describe dentro de una estructura llamada MACHINE:

MACHINE *Bank*
. . . la descripción de la máquina de estados va aquí. . .
END

Como se puede ver la máquina de estados tiene un nombre: *Bank*.

3.1. Variables de estado

Definir las variables de estado es, tal vez, el paso más complejo al escribir una especificación B. En particular determinar el tipo de cada variable de estado es especialmente importante para lograr una especificación clara, simple y concisa. Los tipos tienen que ser *abstractos*; es decir, deben representar un objeto matemático más que un objeto vinculado a la programación. Entonces, muy a menudo en B los tipos de las variables de estado son conjuntos, relaciones binarias y funciones parciales. Los tipos de una especificación B tratan de capturar la *esencia* de la cosa que está siendo modelada. Por ejemplo, si la esencia de una cosa es ser un número entero, entonces el tipo B es $\mathbb{Z}$ en lugar del tipo `int` de un lenguaje de programación. En efecto, si $x \in \mathbb{Z}$ entonces x representa a *cualquier* número entero mientras que si el tipo fuese `int`, x podría representar solo los números enteros en el intervalo [`minint`,`maxint`]. En otras palabras, un `int` ocupa, digamos, cuatro bytes, pero un $\mathbb{Z}$ no ocupa ningún byte porque es un objeto matemático. Si x e y son de tipo `int` entonces $x+y$ podría dar un resultado inesperado debido a desbordes; si su tipo fuese $\mathbb{Z}$, $x+y$ siempre tiene el valor correcto. Cuando escribimos una especificación no queremos preocuparnos por posibles desbordes de tipos.

Otro ejemplo de la diferencia entre un tipo B y un tipo de implementación surge al considerar el nombre de una persona. Si quieren guardar el nombre de una persona en una variable de un lenguaje de programación, entonces probablemente piensen en una cadena de caracteres de alguna longitud apropiada. Más aun, probablemente analizarían si 40 caracteres es demasiado poco y si 120 serían demasiados. Por el contrario, en B se debería usar un tipo básico:

NAME

donde uno puede almacenar nombres de cualquier longitud porque en una especificación B el uso de la memoria no importa en lo más mínimo. Es decir, en una especificación B no pensamos en términos de la memoria necesaria porque vamos a pensar en eso cuando hayamos entendido qué es lo que tenemos que implementar. Para entender qué es lo que tenemos que implementar escribimos la especificación B. De hecho, si tenemos $n \in NAME$ ni siquiera podemos hablar del tamaño o longitud de n: no tiene tamaño porque pertenece a un tipo básico del cual no sabemos nada sobre la estructura o forma de sus elementos.

Pongamos atención ahora en los tipos que necesitamos para especificar las cajas de ahorros. Como dijimos, los clientes se identifican mediante su número de documento de identidad y estos son usados para identificar sus cajas de ahorros. Por lo tanto, estos números de identidad son importantes en los requerimientos. Sin embargo, ¿los números de identidad son realmente números? ¿Son cadenas de caracteres? ¿Se componen de más de un valor? En realidad no importa. Lo único que importa es que los números de identidad identifican a una persona y su caja de ahorros. En consecuencia, declaramos un tipo básico de nombre *NIC* (por 'national identity card') dentro de la máquina *Bank* de la siguiente forma:

MACHINE *Bank*
SETS *NIC*
  . . . la descripción de la máquina de estados continúa aquí. . .
END

De esta forma no sabemos nada sobre la estructura de los elementos de *NIC*. No obstante, B nos permite construir conjuntos de *NIC* y definir relaciones binarias y funciones parciales que toman o devuelven *NIC*. Además, podemos usar los operadores matemáticos básicos sobre *NIC*. Por ejemplo, si $n, m \in NIC$; $A, B \in \mathbb{P} NIC$:

$n = m$
$n \neq m$
$n \in A$
$A \subseteq B$

son todos predicados B.

Es muy importante que observen que $n = m$ es un predicado, no una asignación. Por este motivo podemos escribir $n = m$ o $m = n$ o $\neg (n \neq m)$.

Los requerimientos dicen que necesitamos mantener el saldo de cada caja de ahorros y que es posible depositar y retirar cantidades de dinero. Luego necesitamos poder predicar sobre cantidades de dinero en nuestra especificación B. A primera vista una cantidad de dinero parece un número real. Pero, por otro lado, sabemos que las cantidades de dinero no tienen más que dos decimales. Entonces, en realidad, una cantidad de dinero se parece más a un número racional (dado que siempre tiene un número finito de cifras decimales). Más aun, al multiplicar cualquier cantidad de dinero por 100 obtenemos un número entero (de centavos). Por ejemplo, podemos decir que $\$1,32$ equivalen a 132 centavos. En consecuencia, podemos modelar cualquier cantidad de dinero con los números enteros.

Ahora necesitamos pensar en las variables de estado de nuestra especificación. Claramente, el banco va a tener un cierto número de cajas de ahorros en cada momento. Por ejemplo, podemos pensar que el banco no tiene ningún cliente, y en consecuencia ninguna caja de ahorros, cuando abre al público por primera vez. Luego de que el primer cliente abre su caja de ahorros el banco pasa de cero cajas de ahorro a una cuyo saldo es cero. De esta forma el banco transicionó de un estado donde no tenía ninguna caja de ahorros a un estado donde hay solo una caja de ahorros y su saldo es cero. Ahora el titular de la cuenta puede depositar \$200, lo que hace que el banco transicione a un nuevo estado donde hay solo una caja de ahorros pero ahora su saldo es 200. Más tarde un nuevo cliente abre su caja de ahorros haciendo que el banco transicione a un nuevo estado donde hay dos cajas de ahorro, una con saldo 200 y la otra con saldo cero.

En resumen: *cada estado del banco se caracteriza por el número de cajas de ahorro, la identidad de cada cliente y el saldo de cada caja de ahorros*.

Los requerimientos dicen que cada caja de ahorros debe ser identificada con el *NIC* del titular y que se debe mantener el saldo de la caja de ahorros. Entonces podemos representar una caja de ahorros con un par ordenado $(nic, balance)$ donde *nic* es el *NIC* del titular de la cuenta y *balance* es el saldo actual de la cuenta. Por lo tanto, podemos poner todos estos pares ordenados en un conjunto para tener el estado del banco:

$$\{(nic_1, balance_1), \dots, (nic_n, balance_n)\}$$

Es decir, tenemos *el número de cajas de ahorros, la identidad de cada titular y el saldo de cada cuenta*. De esta forma si se depositan 120 pesos en la cuenta identificada con nic_1 tenemos un nuevo estado:

$$\{(nic_1, balance_1 + 120), \dots, (nic_n, balance_n)\}$$

En consecuencia, podemos especificar el estado del banco con una única variable de estado cuyos valores son conjuntos de pares ordenados. Vamos a llamar a esta variable *sa* (por *savings accounts*). Ahora establecemos que *sa* es la variable de estado de la máquina *Bank*, de la siguiente forma:

Machine *Bank*
Sets *NIC*
Variables *sa*
. . . la descripción de la máquina de estados continúa aquí. . .
End

Todo conjunto de pares ordenados es una relación binaria. En nuestro caso el dominio de la relación binaria es *NIC* y el rango es $\mathbb{N}$, porque los saldos de las cuentas no pueden ser negativos. Más aun, observen que todos los nic_i son diferentes ente sí dado que cada persona tiene un *NIC* distinto de todos los demás y los requerimientos dicen que cada persona puede tener a lo sumo una caja de ahorros. Entonces, *para cada* nic_i *hay un único saldo*. Luego, en una segunda aproximación, tenemos $sa \in NIC \longrightarrow \mathbb{N}$. Es decir, *sa* es una función. Sin embargo, no todas las personas van a ser clientes del banco todo el tiempo. Esto significa que puede existir $n \in NIC$ para el cual $sa(n)$ no está definida porque la persona cuyo *NIC* es *n* no es un cliente del banco. Entonces, finalmente, llegamos al tipo correcto para la variable de estado, $sa \in NIC \nrightarrow \mathbb{N}$. Es decir, *sa* es una función parcial porque no toda persona es un cliente del banco todo el tiempo. El tipo de *sa* se especifica de la siguiente forma en la máquina de estados *Bank*:

Machine *Bank*
Sets *NIC*
Variables *sa*
Invariant $sa \in NIC \nrightarrow \mathbb{N}$
. . . la descripción de la máquina de estados continúa aquí. . .
End

La palabra Invariant indica que $sa \in NIC \nrightarrow \mathbb{N}$ es o debería ser una *invariante de estado*. Una invariante de estado es un predicado que es verdadero en todos los estados de la máquina.

Entonces, estamos diciendo que queremos que *sa* sea una función parcial de *NIC* en $\mathbb{N}$ en todos los estados del banco. Claramente, si llegamos a un estado del banco donde hay un par ordenado $(n,b) \in sa$ tal que $b < 0$ estamos en problemas; de la misma forma, si el banco alguna vez llega a un estado tal que $(n,b_1),(n,b_2) \in sa \land b_1 \neq b_2$, estamos en problemas. ¿Cómo podemos estar seguros de que el banco nunca va a llegar a uno de esos estados? La respuesta la veremos en la Sección 5. Por ahora es suficiente con saber que Invariant se usa para indicar el tipo de las variables de estado.

Como cualquier máquina de estados *Bank* inicia su ejecución en algún estado inicial. En este caso el estado inicial es cuando no hay ninguna caja de ahorros abierta en el banco. Lo codificamos en B de la siguiente forma:

Machine *Bank*
Sets *NIC*
Variables *sa*
Invariant $sa \in NIC \nrightarrow \mathbb{N}$
Initialisation $sa := \{\}$
. . . la descripción de la máquina de estados continúa aquí. . .
End

Recuerden que en B las funciones parciales son conjuntos de pares ordenados. Entonces es correcto decir que una función parcial es igual al conjunto vacío.

También observen que usamos :=, llamado *asignación abstracta*, para establecer el estado inicial. La asignación abstracta se usa para establecer el valor de las variables de estado y los parámetros de salida. $x := expr$ puede leerse como "x toma el valor $expr$". En $x := expr$, x debe ser una variable de estado o un parámetro de salida y $expr$ puede ser una expresión.

3.1.1. ¿Por qué no usar una lista en lugar de una función para *sa*?

¿Qué pasa si *sa* es una lista de pares ordenados en lugar de una función parcial? Por ejemplo, en Java podemos definir la clase `SavingsAccount` con dos variables miembro, `nic` y `bal`, para almacenar el *NIC* y el saldo de una caja de ahorros. Entonces podemos definir una lista de `SavingsAccount`:

```
List<SavingsAccount> sa;
```

donde podemos almacenar el estado completo del banco.

¿Por qué no hacer lo mismo en B? De hecho B provee listas como conjuntos de pares ordenados, llamados secuencias. Entonces, podríamos haber definido *sa* como $sa \in \text{seq}(NIC \times \mathbb{N})$ en lugar de como una función parcial. ¿Por qué no usamos este tipo el cual, después de todo, es más cercano a una implementación Java? La respuesta es: porque *sa* no exactamente una secuencia. Por ejemplo, el siguiente es un valor posible para *sa* si la declaramos como una secuencia:

$$[(nic_1,100),(nic_2,230),(nic_1,529)]$$

Es decir, aparentemente, el banco tiene tres cajas de ahorro. Sin embargo, observen que el primer y el último par tienen el mismo identificador de cuenta, nic_1, pero tienen saldos diferentes, 100 y 529. ¿Es correcto esto? ¿Cuál es el saldo correcto de la caja de ahorros nic_1? La respuesta es que

esto no es correcto porque una caja de ahorro debe tener un único saldo. El problema es que una secuencia (o lista) permite repeticiones lo que abre la posibilidad a tener el mismo identificador de cuenta con diferentes saldos. Pero, según los requerimientos, esto no debería ser posible.

Por otro lado, si *sa* se define como una función parcial, el siguiente conjunto:

$$\{(nic_1, 100), (nic_2, 230), (nic_1, 529)\}$$

no puede ser un valor de *sa* porque este conjunto no es una función parcial.

El punto es que una función parcial modela correctamente la esencia de *sa*. No tiene importancia si las funciones parciales no están disponibles en los lenguajes de programación o si no son lo suficientemente eficientes, porque estamos escribiendo una *especificación* formal, no estamos escribiendo un programa[1]. Escribimos una especificación porque queremos entender claramente lo que el programa tiene que hacer. Por el momento no estamos interesados en las cuestiones de desempeño; por ahora no nos interesa proveer un programa eficiente, solo nos interesa proveer un programa *correcto*. Más tarde podemos (y lo haremos) prestar atención a la cuestión de escribir un programa *correcto y eficiente*.

En otras palabras:

- Cuando escribimos una especificación estamos definiendo el *problema*
- Cuando escribimos un programa estamos definiendo la *solución* de ese problema

3.2. Especificación de las operaciones sobre las cajas de ahorros

Hasta el momento la especificación es la siguiente:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \mathrel{+\!\!\!\rightarrow} \mathbb{N}$
INITIALISATION $sa := \{\}$
... la descripción de la máquina de estados continúa aquí...
END

Es decir, tenemos los tipos que estamos usando en la especificación, la definición del conjunto de estados del banco y el estado inicial.

El siguiente paso es especificar las operaciones que pueden ser ejecutadas sobre el banco. Según los requerimientos debemos especificar las siguiente cinco operaciones (que son clasificadas en transiciones de estado o consultas de estado):

1. Abrir una caja de ahorros (transición de estados)
2. Depositar dinero en una caja de ahorros (transición de estados)
3. Retinar dinero de una caja de ahorros (transición de estados)
4. Consultar el saldo disponible (consulta de estado)
5. Cerrar una caja de ahorros (transición de estados)

[1] De hecho, las funciones parciales están disponibles en, por caso, Java como tablas *hash*.

3.2.1. Abrir una caja de ahorros (transición de estados)

La operación que especifica cómo se abre una caja de ahorros se llama **open**. Para abrir una cuenta necesitamos el número de cuenta con el que se va a identificar esta nueva cuenta. Este dato es un *parámetro de entrada* de **open**; el nombre de este parámetro será *an*[2]. Las operaciones se especifican en la sección Operations, como se indica a continuación.

Machine *Bank*
Sets *NIC*
Variables *sa*
Invariant $sa \in NIC \rightarrowtail \mathbb{N}$
Initialisation $sa := \{\}$
Operations
  open(*an*) $\widehat{=}$ ... *especificar la operación aquí...*
    ... *la descripción de la máquina de estados continúa aquí...*
End

Como dijimos 'abrir una caja de ahorros' es una transición de estados que se puede describir de la siguiente forma:

- Tomar *an*
- Controlar si $an \in NIC$. De lo contrario **open** podría ser invocada con un valor para *an* que no es un identificador de cuenta.
- Controlar si hay una caja de ahorros cuyo identificador es *an* (pues no podemos abrir la misma cuenta dos veces).

  Todos los identificadores de cuentas usados en el banco son las primeras componentes de los pares ordenados guardados en *sa*. Este conjunto se puede obtener fácilmente calculando el dominio de *sa*, el cual es exactamente eso: el conjunto de las primeras componentes de los pares ordenados de *sa*.

  El dominio de *sa* se escribe $\mathrm{dom}(sa)$. Entonces, controlar que no haya una caja de ahorros cuyo identificador sea *an* se puede escribir como $an \notin \mathrm{dom}(sa)$.
- Si *an* identifica una nueva caja de ahorros, entonces hay que modificar el estado actual del banco agregando un par ordenado de la forma $(an, 0)$ a *sa*.

  Agregamos $(an, 0)$ porque, según los requerimientos, el saldo de la nueva cuenta es cero.

  Se puede agregar el par ordenado $(an, 0)$ a *sa* mediante unión de conjuntos: $sa \cup \{(an, 0)\}$. Es decir, $sa \cup \{(an, 0)\}$ es una función parcial igual a *sa* salvo que tiene un par ordenado más, $(an, 0)$.

  Observen que escribimos $sa \cup \{(an, 0)\}$ y no $sa \cup (an, 0)$ porque $\cup$ espera dos conjuntos ($(an, 0)$ es un par ordenado, no un conjunto; $\{(an, 0)\}$ es un conjunto unitario cuyo elemento es un par ordenado).

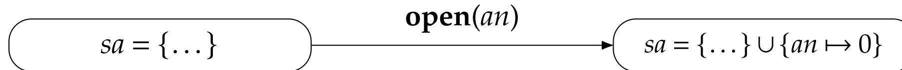
  En B el par ordenado (x, y) se puede escribir también como $x \mapsto y$. Por ejemplo, $sa \cup \{an \mapsto 0\}$. $x \mapsto y$ se puede leer como "la imagen de x es y". De ahora en más vamos a usar la notación basada en $\mapsto$.

[2]En la Sección 9 vamos a especificar una versión de la operación donde el sistema genera el número de cuenta y por lo tanto este parámetro de entrada no será necesario.

Ahora escribimos todo lo anterior en la máquina como se muestra a continuación:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALISATION $sa := \{\}$
OPERATIONS
open(an) $\widehat{=}$
PRE $an \in NIC \land an \notin \mathrm{dom}(sa)$
THEN $sa := sa \cup \{an \mapsto 0\}$
END;
. . . la descripción de la máquina de estados continúa aquí. . .
END

La especificación de **open** establece que si el predicado escrito en la sección **PRE** vale, entonces el predicado escrito en la sección **THEN** debe ser verdadero. Como ocurre en la sección INITIALISATION usamos una asignación abstracta para establecer el valor de la variable de estado en el nuevo estado. La asignación abstracta es una simplificación sintáctica que representa un predicado como veremos en la Sección 5. De esta forma, la especificación de **open** se puede graficar de la siguiente forma:



Es decir, si se ejecuta **open**(an) desde un estado donde sa es igual a cierto conjunto, indicado con $\{\dots\}$, entonces el valor de sa en el estado siguiente es $\{\dots\} \cup \{(an, 0)\}$.

En general, una operación se especifica dando sus *precondiciones* y sus *postcondiciones*. La sección **PRE** solo puede contener precondiciones; la sección **THEN**, llamada *cuerpo*, puede contener precondiciones y postcondiciones. Entonces, en **open** la precondición es $an \in NIC \land an \notin \mathrm{dom}(sa)$, y la postcondición es $sa := sa \cup \{an \mapsto 0\}$. Una precondición es un predicado que se *espera* sea verdadero *antes* de que la operación sea ejecutada y una postcondición es un predicado que *es* verdadero *después* que la operación ha sido ejecutada. En otras palabras, la especificación dice que si la precondición es verdadera antes de que la operación sea invocada, entonces la postcondición será verdadera luego de que la operación haya sido ejecutada. La especificación no dice nada sobre el comportamiento de la máquina cuando la precondición de la operación que es invocada no vale.

En general, una precondición es un predicado que depende unicamente de variables de estado y parámetros de entrada y no incluye asignaciones abstractas; y una postcondición es un predicado que depende de cualquier variable (estado, entrada y salida) y puede incluir asignaciones abstractas. Por ejemplo, $an \in NIC \land an \notin \mathrm{dom}(sa)$ es una precondición porque depende sólo de an (que es un parámetro de entrada) y sa (que es una variable de estado); y $sa := sa \cup \{an \mapsto 0\}$ no es una precondición, y por lo tanto es una postcondición, porque usa la asignación abstracta.

Por consiguiente, cuando tienen que especificar una operación comiencen pensando en términos de precondiciones y postcondiciones. Luego, escríbanlas usando el lenguaje B.

Entre las precondiciones debemos dar el tipo de cada parámetro de entrada de la operación. Nosotros hicimos eso escribiendo $an \in NIC$.

Si la precondición es verdadera antes de invocar la operación, entonces la postcondición será verdadera cuando la ejecución de la operación termine.

No hay ningún comportamiento definido si la precondición no se cumple.

El invocante es responsable de llamar a la operación cuando la precondición es verdadera.

Agregando mensajes de error a open. La especificación de la operación que abre una caja de ahorros dada en **open** está bien pero no dice qué debe hacer el sistema si **open** se invoca cuando la precondición no vale. Es decir, ¿qué debe ocurrir si tratamos de abrir una caja de ahorros cuyo identificador ya está siendo usado? Hasta el momento, hemos especificado solo el *comportamiento normal* o el *comportamiento feliz*[3]. Es decir, hemos especificado la operación cuando el invocador no comete errores. Si la operación se llama cuando la precondición no vale, la especificación no indica ningún comportamiento para **open**. Sin embargo, esto no es muy informativo para el usuario o incluso puede ser confuso. Imaginen un usuario que llama a **open** con un número de cuenta que ya está en uso y el sistema no hace nada, no responde nada. El usuario puede pensar que la cuenta fue abierta porque el sistema no muestra un error. Por este motivo vamos a mejorar **open** agregando *mensajes de error* o *códigos de error*. En términos generales vamos a especificar el *comportamiento anormal* o el *comportamiento erróneo*.

Entonces necesitamos decir cuáles son los mensajes que el sistema va a emitir cuando ejecutamos las operaciones. Más aun, necesitamos definir un tipo para los mensajes. ¿Los mensajes de error son cadenas de caracteres? ¿Son más bien códigos de error como por ejemplo el 1 representa la ejecución exitosa y el 0 una ejecución errónea? Como lo hicimos con otros tipos que usamos en la especificación trataremos de abstraer la representación para capturar la esencia de lo que queremos decir. Por este motivo vamos a pensar en los mensajes de error como constante de un tipo enumerado. Por el momento vamos a considerar solo los dos mensajes siguientes:

1. *ok*, significa que la operación ejecutó exitosamente
2. *nicExists*, significa que el identificador de la caja de ahorros ya está en uso en el banco

El hecho de que estos mensajes se parezcan a cadenas de caracteres no significa que lo sean. *ok* y *nicExists* son simplemente valores (o elementos) de un tipo que vamos a definir. Dado que este tipo va a tener un número finito de elementos podemos usar un tipo enumerado. Por lo tanto, definimos lo siguiente:

[3]'happy path', en inglés

MACHINE *Bank*
SETS NIC; $MSG = \{ok, nicExists\}$ [conjunto nuevo]
VARIABLES sa
INVARIANT $sa \in NIC \rightarrow\!\!\!\!\!\!+ \ \mathbb{N}$
INITIALISATION $sa := \{\}$
OPERATIONS
open(an) $\widehat{=}$
PRE $an \in NIC \land an \notin \mathrm{dom}(sa)$
THEN $sa := sa \cup \{an \mapsto 0\}$
END;
. . . la descripción de la máquina de estados continúa aquí. . .
END

Vamos a ir agregando elementos a MSG a medida que los necesitemos.

Para que el sistema pueda mostrar un mensaje de error debe poder emitir una salida. B provee un mecanismo general para que las operaciones produzcan o emitan salidas (pueden ser mensajes de error o cualquier otra información que deba ser enviada al entorno). Este mecanismo se basa en *parámetros de salida*. Un parámetro de salida se declara de la siguiente forma:

$msg \leftarrow$ **open**(an) $\widehat{=} \dots$

De esta forma msg es un parámetro de salida de **open**.

A continuación vamos a redefinir **open** de manera tal que emita ok cuando todo va bien, y $nicExists$ cuando un identificador de cuenta en uso se pasa como parámetro de entrada. Si $an \notin NIC$ la operación no será ejecutada. Dado que la operación produce salida cuando es ejecutada, vamos a tener que modificar la sección **THEN** y mover algunas precondiciones.

MACHINE *Bank*
SETS NIC; $MSG = \{ok, nicExists\}$
VARIABLES sa
INVARIANT $sa \in NIC \rightarrow\!\!\!\!\!\!+ \ \mathbb{N}$
INITIALISATION $sa := \{\}$
OPERATIONS
$msg \leftarrow$ **open**(an) $\widehat{=}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;
. . . la descripción de la máquina de estados continúa aquí. . .
END

Observen la sentencia:

$sa, msg := sa \cup \{an \mapsto 0\}, ok$

Esto se llama *asignación múltiple*. El significado de una asignación múltiple es que se asignan varias variables simultáneamente. Este tipo de asignaciones se puede escribir usando *asignaciones*

paralelas:

$$sa := sa \cup \{an \mapsto 0\} \parallel msg := ok$$

La asignación paralela es conmutativa:

$$x_1 := expr_1 \parallel x_2 := expr_2 \equiv x_2 := expr_2 \parallel x_1 := expr_1$$

Esto deja en claro que las asignaciones no se ejecutan en ningún orden en particular.

También observen que en la rama **ELSE** no cambiamos el estado de la máquina. Entonces cuando $an \in \operatorname{dom}(sa)$ solo se produce una salida.

Finalmente, recuerden que si $an \in NIC$ no es verdadero la operación no se ejecuta; es decir, no hay cambio de estado ni salidas.

¿Por qué no consideramos la posibilidad de una llamada como **open**(*an*) con $an \notin NIC$? En otras palabras, ¿por qué no mostramos un mensaje de error cuando $an \notin NIC$? La razón es la misma que cuando no consideramos que un programa puede ser invocado con un parámetro de entrada que no sea del tipo esperado. La idea es que el programador implemente **open** y *NIC* de forma tal que el programa no pueda ser invocado si se llama a **open** con un valor de *an* que no es del tipo esperado.

Especificaciones e implementaciones. Si esta fuese la especificación completa del sistema de cajas de ahorros deberíamos entregarla a un programador para que la implemente con alguna tecnología de implementación. No sería necesario acompañar la especificación con los requerimientos (Sección 2). La especificación contiene suficiente información como para que el programador pueda implementar un programa correcto. Mejor aun, la especificación es *precisa*, *sin ambigüedades* y *abstracta*. Por lo tanto, la especificación no prohíbe ninguna implementación particular en tanto respete la especificación.

Sin embargo, antes de entregar la especificación al programador podríamos comprobar *matemáticamente* que la especificación es más o menos correcta. Veremos más sobre esto en la Sección 5. Más aun, las especificaciones formales pueden ser transformadas *matemáticamente* en programas. De esta forma, se dice que la implementación es *correcta por construcción*. El método B fue especialmente diseñado para transformar especificaciones en programas siguiendo reglas matemáticas, aunque no veremos cómo hacer eso en este curso.

3.2.2. Depositar dinero en una caja de ahorros (transición de estados)

Una vez que tenemos una caja de ahorros podemos depositar dinero en ella. La operación, llamada **deposit**, recibe dos entradas: el identificador de la cuenta, *an*, y el monto de dinero a ser depositado, *amt*. El tipo de *an* es *NIC* y el tipo de *amt* es $\mathbb{N}$. Las precondiciones para depositar una cantidad de dinero *amt* en la caja de ahorros *an* son las siguientes:

1. *an* debe ser un *NIC*. Formalmente:

   $$an \in NIC$$

   Este es el tipo de *an*. Es obligatorio declarar el tipo de cada parámetro de entrada.

2. *an* debe ser un identificador de cuenta válido. Formalmente:

   $$an \in \operatorname{dom}(sa)$$

3. *amt* debe ser un número natural. Formalmente:

   $$amt \in \mathbb{N}$$

   Este es el tipo de *amt*. Es obligatorio declarar el tipo de cada parámetro de entrada.

4. *amt* debe ser un número positivo (no tiene sentido depositar cero pesos). Formalmente:

   $$amt > 0$$

Las postcondiciones del *comportamiento normal* son las siguientes:

1. Se debe emitir el mensaje *ok*.

2. El saldo de *an* debe ser igual al saldo en el estado de partida más *amt*; todas las otras cajas de ahorros permanecen sin cambios; no se agregan ni se eliminan cuentas del banco.

   El saldo de *an* en el estado de partida es $sa(an)$. En efecto, dado que *sa* es una función de *NIC* a $\mathbb{N}$, entonces $sa(an)$ es el saldo de *an*. Es lo mismo que preguntar, ¿cuál es el coseno de π? La respuesta es $\cos(\pi)$. Es decir, solo aplicamos la función a su argumento. Además, *sa* denota el conjunto de cajas de ahorros del banco en el estado de partida, por lo que $sa(an)$ es el saldo de *an* en el estado de partida. Ahora, $sa(an) + amt$ es igual al saldo de *an* en el estado de partida más *amt*. Entonces, $sa(an) + amt$ debe ser el saldo de *an* en el estado de llegada.

   Ahora miramos las otras condiciones: todas las otras cajas de ahorros permanecen sin cambios; no se agregan ni se eliminan cuentas del banco.

   Resumiendo, tenemos que modificar *sa* solo en un punto (*an*) y dejar el resto de la función tal cual está.

   Como estas condiciones aparecen con mucha frecuencia en especificaciones B, el lenguaje provee un operador, llamado *actualizar* notado $\Leftarrow\!\!+$, que especifica esas tres condiciones en un predicado atómico.

   Entonces, la asignación abstracta correcta en B para dar esas tres condiciones es la siguiente:

   $$sa := sa \Leftarrow\!\!+ \{an \mapsto sa(an) + amt\}$$

   Actualizar una función significa modificar la segunda componente de uno o más de sus pares ordenados pero también significa agregar pares ordenados nuevos a la función. En consecuencia, en general podemos usar $\Leftarrow\!\!+$ de la siguiente forma:

   $$f \Leftarrow\!\!+ \{(x_1, y_1), \dots, (x_n, y_n)\}$$

   Si un x_i no pertenece al dominio de f entonces $\Leftarrow\!\!+$ agrega el par ordenado (x_i, y_i) a f. Entonces, la función dada por $f \Leftarrow\!\!+ \{(x_1, y_1), \dots, (x_n, y_n)\}$ siempre va a estar definida en todos los x_i.

   Los detalles matemáticos de $\Leftarrow\!\!+$ se introducen más adelante.

Ahora podemos especificar el comportamiento normal cuando se hace un depósito.

$$
\begin{array}{l}
\textsc{Machine}\ \ Bank \\
\textsc{Sets}\ \ NIC;\ MSG = \{ok, nicExists\} \\
\textsc{Variables}\ \ sa \\
\textsc{Invariant}\ \ sa \in NIC \nrightarrow \mathbb{N} \\
\textsc{Initialisation}\ \ sa := \{\} \\
\textsc{Operations} \\
\quad msg \longleftarrow \mathbf{open}(an) \mathrel{\widehat=} \\
\quad\ \ \textbf{PRE}\ \ an \in NIC \\
\quad\ \ \textbf{THEN}\ \ \textbf{IF}\ \ an \notin \mathrm{dom}(sa)\ \ \textbf{THEN}\ \ sa, msg := sa \cup \{an \mapsto 0\}, ok\ \ \textbf{ELSE}\ \ msg := nicExists\ \ \textbf{END} \\
\quad\ \ \textbf{END}; \\
\quad msg \longleftarrow \mathbf{deposit}(an, amt) \mathrel{\widehat=} \\
\quad\ \ \textbf{PRE}\ \ an \in NIC \land amt \in \mathbb{N} \land an \in \mathrm{dom}(sa) \land 0 < amt \\
\quad\ \ \textbf{THEN}\ \ sa, msg := sa \mathbin{\lhd\!\!-} \{an \mapsto sa(an) + amt\}, ok \\
\quad\ \ \textbf{END}; \\
\quad\ \ \ \ldots \textit{la descripción de la máquina de estados continúa aquí}\ldots \\
\textsc{End}
\end{array}
$$

Aunque $0 < amt$ implica $amt \in \mathbb{N}$, B exige que $amt \in \mathbb{N}$ esté presente in la precondición. Lo mismo ocurre con $an \in \mathrm{dom}(sa)$ y $an \in NIC$. Podemos decir que $amt \in \mathbb{N}$ y $an \in NIC$ son *restricciones de tipo*, mientras que $0 < amt$ y $an \in \mathrm{dom}(sa)$ son *precondiciones funcionales*. Técnicamente ambas clases de predicados son precondiciones pero tienen un origen diferente.

A continuación tenemos que especificar los comportamientos anormales. Cada comportamiento anormal surge de negar una precondición *funcional* y establecer un mensaje de error para cada negación[4]. En el caso de **deposit** tenemos dos comportamientos anormales dados por la negación de cada precondición funcional:

- $an \notin \mathrm{dom}(sa)$ con mensaje de error *nicNotExists*
- $amt = 0$ con mensaje de error *amountError*

En consecuencia tenemos tres comportamientos para **deposit**: uno normal más dos erróneos. Podemos especificar todo esto con un **IF-THEN-ELSE** anidado, como se muestra a continuación:

$$
\begin{array}{l}
\textbf{IF}\ \ an \in \mathrm{dom}(sa) \land 0 < amt \\
\textbf{THEN}\ \ sa, msg := sa \mathbin{\lhd\!\!-} \{an \mapsto sa(an) + amt\}, ok \\
\textbf{ELSE}\ \ \textbf{IF}\ \ an \notin \mathrm{dom}(sa)\ \ \textbf{THEN}\ \ msg := nicNotExists\ \ \textbf{ELSE}\ \ msg := amountError\ \ \textbf{END} \\
\textbf{END}
\end{array}
$$

Sin embargo, B provee una sentencia más conveniente para estos casos que evita el anidamiento:

$$
\begin{array}{l}
\textbf{SELECT}\ \ an \in \mathrm{dom}(sa) \land 0 < amt\ \ \textbf{THEN}\ \ sa, msg := sa \mathbin{\lhd\!\!-} \{an \mapsto sa(an) + amt\}, ok \\
\textbf{WHEN}\ \ an \notin \mathrm{dom}(sa)\ \ \textbf{THEN}\ \ msg := nicNotExists \\
\textbf{WHEN}\ \ amt = 0\ \ \textbf{THEN}\ \ msg := amountError \\
\textbf{END}
\end{array}
$$

[4]Otra opción consiste en definir un solo comportamiento anormal que cubra todos los casos de error. En ese caso se puede establecer un mensaje de error genérico.

```
MACHINE  Bank
SETS  NIC; MSG = {ok, nicExists, nicNotExists, amountError}          [more elements in MSG]
VARIABLES  sa
INVARIANT  sa ∈ NIC ⇸ ℕ
INITIALISATION  sa := {}
OPERATIONS
  msg ← open(an) ≙
    PRE  an ∈ NIC
    THEN  IF  an ∉ dom(sa)  THEN  sa, msg := sa ∪ {an ↦ 0}, ok  ELSE  msg := nicExists  END
    END;
  msg ← deposit(an, amt) ≙
    PRE  an ∈ NIC ∧ amt ∈ ℕ
    THEN
      SELECT  an ∈ dom(sa) ∧ 0 < amt   THEN  sa, msg := sa ⩥ {an ↦ sa(an) + amt}, ok
      WHEN  an ∉ dom(sa)   THEN  msg := nicNotExists
      WHEN  amt = 0   THEN  msg := amountError
      END
    END
END
```

Figura 1: Especificación de la operación **deposit**

Si en una sentencia **SELECT** varias condiciones son verdaderas al mismo tiempo, entonces la selección de la rama a ejecutar es no-determinista. Si ninguna de las ramas es verdadera, entonces la sentencia entera es falsa (o no se ejecuta). En el caso de **deposit** las últimas dos ramas pueden ser verdaderas simultáneamente. Esto significa que en ese caso **deposit** mostrará solo un mensaje de error, pero no sabemos cuál de los dos.

La especificación completa de la operación se puede ver en la Figura 1.

Observen que movimos las precondiciones funcionales a la sección **THEN** con el fin de poder seleccionar la postcondición correcta para cada caso. De hecho, la especificación de **deposit** es equivalente a la siguiente fórmula:

$$
\begin{aligned}
& an \in NIC \land amt \in \mathbb{N} \\
& \land (an \in \mathrm{dom}(sa) \land 0 < amt \Rightarrow sa, msg := sa \oplus \{an \mapsto sa(an) + amt\}, ok) \\
& \land (an \notin \mathrm{dom}(sa) \Rightarrow msg := nicNotExists) \\
& \land (amt = 0 \Rightarrow msg := amountError)
\end{aligned} \tag{1}
$$

Detalles matemáticos del operador actualizar ($\oplus$) y otros operadores importantes. Si X e Y son dos tipos B y $R, S \in X \leftrightarrow Y$, entonces la definición formal del operador actualizar es la siguiente:

$$R \oplus S = ((\mathrm{dom}(S)) \lhd R) \cup S$$

donde, si $A \in \mathbb{P}X$ entonces:

$$A \mathbin{⩤} R = \{x \mapsto y \mid x \mapsto y \in R \land x \notin A\}$$

Por lo tanto, en realidad $\Leftarrow$ está definido para relaciones binarias y no solo para funciones parciales. Dado que toda función parcial es una relación binaria entonces también podemos aplicar $\Leftarrow$ a funciones parciales. Claramente, si $R \in X \leftrightarrow Y$ y $f \in X \mathrel{+\!\!\!\rightarrow} Y$, entonces $R \Leftarrow f$ y $f \Leftarrow R$ tipan correctamente. Además, como cualquier conjunto de la forma $\{x \mapsto y\}$ es una relación binaria entonces podemos escribir $sa \Leftarrow \{an \mapsto sa(an) + amt\}$ como hicimos en **deposit**. Tengan presente, no obstante, que $\Leftarrow$ está definido para dos relaciones binarias del *mismo* tipo.

Como puede verse, $\Leftarrow$ está definido en términos de otro operador B llamado *anti-restricción de dominio* ($⩤$). Este operador toma un conjunto y una relación binaria, y retorna otra relación binaria. En efecto, la relación binaria resultante es una *restricción* de la relación inicial; es decir, $A ⩤ R \subseteq R$. $A ⩤ R$ elimina de R todos los pares ordenados cuya primera componente no pertenece a A. Por ejemplo:

$$\{2,4\} ⩤ \{(5,a),(4,w),(4,q),(7,e),(2,q)\} = \{(5,a),(7,e)\}$$

Por lo tanto, si $x \in \mathrm{dom}(R)$ entonces $R \Leftarrow \{x \mapsto y\}$ primero elimina de R todos los pares ordenados de la forma $x \mapsto \bullet$ y luego agrega $x \mapsto y$ a la relación resultante. En otras palabras:

$$\mathrm{dom}(\{x \mapsto y\}) ⩤ R = \{x\} ⩤ R \qquad \text{[elimina pares de la forma } x \mapsto \cdot\text{]}$$

$$(\{x\} ⩤ R) \cup \{x \mapsto y\} = R \Leftarrow \{x \mapsto y\} \qquad \text{[agrega } x \mapsto y\text{]}$$

Claramente, si $x \notin \mathrm{dom}(R)$ entonces $R \Leftarrow \{(x,y)\}$ es igual a $R \cup \{(x,y)\}$.

Consideren el siguiente ejemplo:

$$\begin{aligned}
&\{(5,a),(4,w),(4,q),(7,e),(2,q)\} \Leftarrow \{(7,x)\} \\
&= (\mathrm{dom}(\{(7,x)\}) ⩤ \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= (\{7\} ⩤ \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= \{(5,a),(4,w),(4,q),(2,q)\} \cup \{(7,x)\} \\
&= \{(5,a),(4,w),(4,q),(2,q),(7,x)\}
\end{aligned}$$

Pero también observen este otro:

$$\begin{aligned}
&\{(5,a),(4,w),(4,q),(7,e),(2,q)\} \Leftarrow \{(4,y)\} \\
&= (\mathrm{dom}(\{(4,y)\}) ⩤ \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(4,y)\} \\
&= (\{4\} ⩤ \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= \{(5,a),(7,e),(2,q)\} \cup \{(4,y)\} \\
&= \{(5,a),(7,e),(2,q),(4,y)\}
\end{aligned}$$

donde la relación binaria resultante tiene menos elementos que la inicial.

B define varios operadores conjuntistas y relacionales muy expresivos tales como $\Leftarrow$ y $⩤$. La siguiente tabla lista algunos de ellos; el libro de texto de B provee la lista completa.

Name	Parameters	Definition
Rango	$R \in X \leftrightarrow Y$	$\mathrm{ran}\, R = \{y \mid y \in Y \land \exists x.(x \in X \land (x,y) \in R)\}$
Restricción de dominio	$R \in X \leftrightarrow Y$; $A \in \mathbb{P}X$	$A \lhd R = \{(x,y) \mid (x,y) \in R \land x \in A\}$
Restricción de rango	$R \in X \leftrightarrow Y$; $B \in \mathbb{P}Y$	$R \rhd B = \{(x,y) \mid (x,y) \in R \land y \in B\}$
Anti-restricción de rango	$R \in X \leftrightarrow Y$; $B \in \mathbb{P}Y$	$R ⩥ B = \{(x,y) \mid (x,y) \in R \land y \notin B\}$
Imagen relacional	$R \in X \leftrightarrow Y$; $A \in \mathbb{P}X$	$R[A] = \mathrm{ran}(A \lhd R)$

> Es mejor escribir fórmulas que usen operadores conjuntistas y relacionales que aquellas que usen cuantificadores existenciales y universales. La colección de operadores provista por B permite expresar muchas especificaciones complejas. La mayoría de las veces se puede evitar el uso de cuantificadores. Cada vez que estén por escribir una fórmula con cuantificadores universales piénsenlo dos veces: la mayoría de las veces hay uno o más operadores B que pueden reemplazar al cuantificador.

Simplificación sintáctica para la actualización de una función. En la postcondición de **deposit** tenemos la asignación abstracta:

$$sa := sa \lhd\!\!\!+ \{an \mapsto sa(an) + amt\}$$

Este tipo de asignaciones se llama *actualización de una función en un punto*. Efectivamente, estamos actualizando la función sa en el punto an con la expresión $sa(an) + amt$. La actualización de funciones en un punto es tan frecuente en B que el lenguaje provee una simplificación sintáctica para la asignación abstracta. La asignación anterior se puede escribir de forma simplificada de la siguiente forma:

$$sa(an) := sa(an) + amt$$

Observen que esta forma de asignación tiene una expresión a la izquierda ($sa(an)$) y no una variable, como lo exige la notación B en general. El punto es que, si f es una función y x y e son expresiones, entonces una asignación de la forma:

$$f(x) := e$$

se puede transformar aplicando una transformación sintáctica en la asignación abstracta:

$$f := f \lhd\!\!\!+ \{x \mapsto e\}$$

De ahora en más siempre que sea posible usaremos esta forma simplificada de la asignación abstracta.

Volvemos ahora a la especificación de las otras operaciones del sistema de cajas de ahorros.

3.2.3. Retinar dinero de una caja de ahorros (transición de estados)

Retirar dinero de una cuenta es casi igual a depositar. La operación, llamada **withdraw**, recibe los mismos dos parámetros de entrada, an y amt. Las precondiciones funcionales son las mismas que en **deposit** más la siguiente:

- amt debe ser menor o igual al saldo actual de an. Formalmente:

  $$amt \leq sa(an)$$

Las postcondiciones del caso normal son similares a las de **deposit** excepto que hay que restar amt al saldo actual. Formalmente:

$$sa := sa \lhd\!\!\!+ \{an \mapsto sa(an) - amt\}$$

$$
\begin{array}{l}
\textsc{Machine}\ \ \mathit{Bank}\\
\textsc{Sets}\ \ \mathit{NIC};\ \mathit{MSG} = \{ok, nicExists, nicNotExists, amountError, insufficientFunds\}\\
\textsc{Variables}\ \ sa\\
\textsc{Invariant}\ \ sa \in NIC \mapsto \mathbb{N}\\
\textsc{Initialisation}\ \ sa := \{\}\\
\textsc{Operations}\\
\quad msg \longleftarrow \mathbf{open}(an) \mathrel{\widehat{=}}\\
\qquad \mathbf{PRE}\ \ an \in NIC\\
\qquad \mathbf{THEN}\ \ \mathbf{IF}\ \ an \notin \mathrm{dom}(sa)\ \ \mathbf{THEN}\ \ sa, msg := sa \cup \{an \mapsto 0\}, ok\ \ \mathbf{ELSE}\ \ msg := nicExists\ \ \mathbf{END}\\
\qquad \mathbf{END};\\
\quad msg \longleftarrow \mathbf{deposit}(an, amt) \mathrel{\widehat{=}}\\
\qquad \mathbf{PRE}\ \ an \in NIC \land amt \in \mathbb{N}\\
\qquad \mathbf{THEN}\\
\qquad\quad \mathbf{SELECT}\ \ an \in \mathrm{dom}(sa) \land 0 < amt\ \ \mathbf{THEN}\ \ sa(an), msg := sa(an) + amt, ok\\
\qquad\quad \mathbf{WHEN}\ \ an \notin \mathrm{dom}(sa)\ \ \mathbf{THEN}\ \ msg := nicNotExists\\
\qquad\quad \mathbf{WHEN}\ \ amt = 0\ \ \mathbf{THEN}\ \ msg := amountError\\
\qquad\quad \mathbf{END}\\
\qquad \mathbf{END};\\
\quad msg \longleftarrow \mathbf{withdraw}(an, amt) \mathrel{\widehat{=}}\\
\qquad \mathbf{PRE}\ \ an \in NIC \land amt \in \mathbb{N}\\
\qquad \mathbf{THEN}\\
\qquad\quad \mathbf{SELECT}\ \ an \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an)\ \mathbf{THEN}\ \ sa(an), msg := sa(an) - amt, ok\\
\qquad\quad \mathbf{WHEN}\ \ an \notin \mathrm{dom}(sa)\ \ \mathbf{THEN}\ \ msg := nicNotExists\\
\qquad\quad \mathbf{WHEN}\ \ amt = 0\ \ \mathbf{THEN}\ \ msg := amountError\\
\qquad\quad \mathbf{WHEN}\ \ an \in \mathrm{dom}(sa) \land sa(an) < amt\ \ \mathbf{THEN}\ \ msg := insufficientFunds\\
\qquad\quad \mathbf{END}\\
\qquad \mathbf{END}\\
\textsc{End}
\end{array}
$$

Figura 2: Especificación de la operación **withdraw**

lo que, como acabamos de ver, se puede simplificar a:

$$sa(an) := sa(an) - amt$$

Con respecto a los comportamientos anormales tenemos que agregar uno más a los que tenemos en **deposit** debido a la nueva precondición funcional. La especificación completa de **withdraw** se puede ver en la Figura 2.

Evaluación de funciones parciales fuera de su dominio. Observen que en la última rama de la sentencia SELECT de **withdraw** agregamos $an \in \mathrm{dom}(sa)$ como una precondición a la asignación $msg := insufficientFunds$. En general la precondición de un comportamiento anormal es solo la negación de la precondición que estamos considerando. Sin embargo, en esa rama

tenemos la negación de $an \leq sa(an)$ y $an \in \text{dom}(sa)$. El motivo por el cual agregamos $an \in \text{dom}(sa)$ como una precondición es que necesitamos asegurar que $sa(an)$ está definido porque de otro modo el predicado $sa(an) < amt$ tendrá un valor indefinido. En otras palabras, $sa(an) < amt$ es verdadero o falso si $sa(an)$ es un número y $sa(an)$ es un número si an pertenece al dominio de sa. Si $an \notin \text{dom}(sa)$, ¿cuál es el valor de $sa(an)$? En otros términos, si $an \notin \text{dom}(sa)$, ¿cuál es el saldo de an según sa? Si $an \notin \text{dom}(sa)$ significa que an no es una cuenta de sa entonces, ¿podemos decir cuál es su saldo? Por lo tanto, para estar seguros de que $sa(an) < amt$ es un predicado bien definido debemos asegurar que $sa(an)$ está definido por lo que es necesario exigir que $an \in \text{dom}(sa)$.

La misma situación aparece en matemática. Por ejemplo, la división en los números reales, es decir x/y con $x, y \in \mathbb{R}$, está definida solo si $y \neq 0$. Entonces / pude definirse como una función (total):

$$_/_ : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \to \mathbb{R}$$

o como una función parcial:

$$_/_ : \mathbb{R} \times \mathbb{R} \nrightarrow \mathbb{R}$$

Así, por ejemplo, la expresión:

$$\frac{1}{x^2 - 4}$$

es un número real si $x^2 - 4 \neq 0$. Si $x = 2$ la expresión se reduce a $1/0$ la cual es indefinida porque $(1, 0) \notin \text{dom}(/)$ (observen que el dominio de / son pares ordenados). En matemática este problema se soluciona estableciendo que $x/0$ es igual a ∞. Sin embargo, $\infty \notin \mathbb{R}$, es decir ∞ no es un número real. Por lo tanto, en matemática la función se 'totaliza' extendiendo el rango:

$$_/_ : \mathbb{R} \times \mathbb{R} \to (\mathbb{R} \cup \{\infty\})$$

donde $\mathbb{R} \cup \{\infty\}$ a veces se escribe como $\mathbb{R}^\infty$.

De hecho, la misma situación también se da en programación. Cada vez que nos olvidamos de controlar si el denominador de una división no es cero el programa se cuelga (*crashea*). En programación ∞ corresponde a terminación anormal del programa (por ejemplo, con un mensaje tal como `division by zero` o `null pointer exception` o `segmentation fault`). La programación está repleta de funciones parciales. Por ejemplo, los arreglos se pueden ver como funciones parciales. En efecto, si `a` es un arreglo de 20 componentes, entonces intentar acceder a `a[35]` no está definido y puede hacer que el programa se cuelgue. Es decir, una expresión como `a[_]` espera un número entero pero está definida solo si este pertenece al rango de los índices del arreglo ($[1, 20]$ en nuestro ejemplo).

B está repleto de funciones parciales porque también la programación está repleta de funciones parciales. En B resolvemos el problema de evaluar una función parcial definiendo dos comportamientos:

1. Conjugamos $x \in \text{dom}(f)$ a cualquier predicado donde incluimos $f(x)$, para toda función parcial f y toda expresión x; y luego
2. Especificamos otro comportamiento para el caso $x \notin \text{dom}(f)$. Por ejemplo, establecemos que el sistema no transiciona a un nuevo estado y muestra un mensaje de error.

3.2.4. Consultar el saldo disponible (consulta de estado)

Esta es la primera operación de nuestro sistema que no es una transición de estado. Aun así, en B transiciones de estado y consultas de estado son muy similares. La única diferencia significativa es que las consultas de estado no usan la asignación abstracta para asignar variables de estado.

La operación para consultar el saldo actual de una caja de ahorros, llamada **checkBalance**, recibe el identificador de cuenta, an, y emite el saldo en bal. La precondición funcional del comportamiento normal es que an debe ser un identificador válido, i.e. $an \in \mathrm{dom}(sa)$. Hay dos postcondiciones para el comportamiento normal:

1. $bal := sa(an)$, es decir la salida es el saldo actual de an en el estado de partida.
2. $msg := ok$, es decir la operación se ejecutó satisfactoriamente.

Hay solo un comportamiento anormal dado por la negación de la precondición funcional $an \in \mathrm{dom}(sa)$. La operación se muestra en la Figura 3

3.2.5. Cerrar una caja de ahorros (transición de estados)

La última operación que vamos a especificar es el cierre de una caja de ahorros, llamada **close**. Espera el identificador de la cuenta a cerrar, an. Las precondiciones funcionales del comportamiento normal son las siguientes:

1. an debe ser un identificador válido, i.e. $an \in \mathrm{dom}(sa)$.
2. El saldo de an debe ser cero. Formalmente:

$$sa(an) = 0$$

A su vez la postcondición para el comportamiento normal es que el valor de sa en el estado de llegada debe ser igual al del estado de partida menos la caja de ahorros identificada con an. Para sacar la caja de ahorros de sa podemos usar diferencia de conjuntos pero no así:

$$sa := sa \setminus \{an\}$$

porque los tipos no están bien. Recuerden que sa es un conjunto de pares ordenado pero $\{an\}$ no lo es. En cambio, podemos hacer esto:

$$sa := sa \setminus \{an \mapsto 0\}$$

que tipa y es correcto porque sabemos que el saldo actual de an es cero porque es una de las precondiciones. Sin embargo, B ofrece un operador más conveniente para decir lo mismo:

$$sa := \{an\} \lhd sa$$

Recuerden que $\lhd$ se llama anti-restricción de dominio. La expresión $\{an\} \lhd sa$ devuelve una función parcial igual a sa excepto que el par ordenado cuya primera componente es an ha sido eliminado. Esto es exactamente lo que quisimos decir cuando escribimos $sa := sa \setminus \{an\}$, solo que la fórmula basada en $\lhd$ es correcta. En otras palabras, $\{an\} \lhd sa$ elimina de sa el par ordenado cuya primera componente es an sin importar cuál es su segunda componente. Obviamente si $an \notin \mathrm{dom}(sa)$, entonces $\{an\} \lhd sa = sa$.

Por otra parte tenemos dos comportamientos anormales dados por la negación de cada una de las precondiciones funcionales. La especificación completa de **close** se puede ver en la Figura 3.

```
MACHINE  Bank
SETS  NIC;
      MSG = {ok, nicExists, nicNotExists, amountError, insufficientFunds, balanceNotZero}
VARIABLES  sa
INVARIANT  sa ∈ NIC ⇸ ℕ
INITIALISATION  sa := {}
OPERATIONS
  msg ← open(an) ≙
    PRE  an ∈ NIC
    THEN  IF  an ∉ dom(sa)  THEN  sa, msg := sa ∪ {an ↦ 0}, ok  ELSE  msg := nicExists  END
    END;

  msg ← deposit(an, amt) ≙
    PRE  an ∈ NIC ∧ amt ∈ ℕ
    THEN
        SELECT  an ∈ dom(sa) ∧ 0 < amt   THEN  sa(an), msg := sa(an) + amt, ok
        WHEN  an ∉ dom(sa)   THEN  msg := nicNotExists
        WHEN  amt = 0   THEN  msg := amountError
        END
    END;

  msg ← withdraw(an, amt) ≙
    PRE  an ∈ NIC ∧ amt ∈ ℕ
    THEN
        SELECT  an ∈ dom(sa) ∧ 0 < amt ∧ amt ≤ sa(an)
                THEN  sa(an), msg := sa(an) − amt, ok
        WHEN  an ∉ dom(sa)   THEN  msg := nicNotExists
        WHEN  amt = 0   THEN  msg := amountError
        WHEN  an ∈ dom(sa) ∧ sa(an) < amt   THEN  msg := insufficientFunds
        END
    END;

  bal, msg ← checkBalance(an) ≙
    PRE  an ∈ NIC
    THEN  IF  an ∈ dom(sa)  THEN  bal, msg := sa(an), ok  ELSE  msg := nicNotExists  END
    END;

  msg ← close(an) ≙
    PRE  an ∈ NIC ∧ amt ∈ ℕ
    THEN
        SELECT  an ∈ dom(sa) ∧ sa(an) = 0   THEN  sa, msg := {an} ⩤ sa, ok
        WHEN  an ∉ dom(sa)   THEN  msg := nicNotExists
        WHEN  an ∈ dom(sa) ∧ sa(an) ≠ 0   THEN  msg := balanceNotZero
        END
    END
END
```

Figura 3: Especificación de las operaciones **checkBalance** y **close**

4. Parámetros de configuración y sus restricciones

Supongamos que tenemos un nuevo requerimiento para el sistema de cajas de ahorros:

- El banco debe informar al banco central el identificador de cuenta y el monto de todos los depósitos de más de 1 millón de pesos.

Como estamos escribiendo una especificación vamos a abstraer el mecanismo usado para informar al central. Con este fin vamos a especificar una operación que simplemente va a emitir el monto y el identificador cuando se den las condiciones requeridas. Es decir no vamos a especificar si la información se envía a través de un servicio web o en un archivo de texto o vía RPC, etc.

Seguidamente tenemos que pensar en cómo incluir el número 1 millón en la especificación. Podemos pensar que este número sea parte del estado del banco en cuyo caso necesitaríamos definir otra variable de estado. Sin embargo, el banco no fija este número; por el contrario, lo fija un sistema externo. Entonces no tiene mucho sentido que sea parte del estado del banco. De hecho, a nivel de implementación un número como este seguramente estaría guardado en algún archivo de configuración que el sistema accedería al inicio; o, alternativamente, el banco central podría ofrecer una interfaz pública donde los bancos podrían consultar el valor a partir del cual deben informar los depósitos.

Para datos que consideramos externos a nuestro sistema pero que de todas formas necesitamos que nuestro sistema acceda, B ofrece una construcción denominada *parámetros de configuración*. Podemos usar un parámetro de configuración en la máquina *Bank* para disponer del límite establecido por el banco central, de la siguiente forma:

MACHINE *Bank(depRepLim)*
   . . . la descripción de la máquina de estados va aquí. . .
END

De esta forma, *depRepLim* es un parámetro de la máquina *Bank*. A pesar de que el banco central dice que el límite es igual a 1 millón de pesos, sabemos que este número puede cambiar a lo largo del tiempo y que siempre va a ser un número positivo. Podemos especificar esta condición en la sección CONSTRAINTS de la siguiente forma:

MACHINE *Bank(depRepLim)*
CONSTRAINTS $depRepLim \in \mathbb{N} \land 0 < depRepLim$
   . . . la descripción de la máquina de estados continúa aquí. . .
END

Ahora podemos usar *depRepLim* para especificar la operación **deposit** de manera que informe

los depósitos que alcanzan el límite fijado por el banco central.

$rn, ra, msg \longleftarrow \textbf{deposit}(an, amt) \;\widehat{=}$
PRE $an \in NIC \wedge amt \in \mathbb{N}$
THEN
  SELECT $an \in \mathrm{dom}(sa) \wedge 0 < amt$ **THEN**
    IF $depRepLim \leq a?$
    THEN $sa(an), rn, ra, msg := sa(an) + amt, an, amt, ok$
    ELSE $sa(an), msg := sa(an) + amt, ok$
    END
  WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
  WHEN $amt = 0$ **THEN** $msg := amountError$
  END
END

Como dijimos, el informe enviado al banco central se abstrae simplemente emitiendo el número de cuenta (*rn*) y el monto del depósito (*ra*).

Podemos parametrizar una máquina B con todos los parámetros que necesitemos.

MACHINE *MyMachine*($P_1, \ldots, P_n, p_1, \ldots, p_m$)
  . . . la descripción de la máquina de estados va aquí. . .
END

Los parámetros que corresponden a tipos de datos se escriben en mayúscula (i.e. $P_1, \ldots, P_n$); los otros parámetros se escriben en minúscula (i.e. $p_1, \ldots, p_m$). Si la sección CONSTRAINTS está presente va inmediatamente debajo de la línea MACHINE. CONSTRAINTS se usa para describir condiciones o restricciones sobre los parámetros de configuración. Los parámetros de configuración se pueden usar en las secciones INVARIANT y OPERATIONS.

Los parámetros de configuración se pueden usar para *subespecificar* ciertos aspectos del sistema. En este contexto, subespecificar significa escribir una especificación más abstracta de lo habitual o una especificación con aun menos detalles. La subespecificación se origina principalmente por tres razones:

1. Al momento de escribir la especificación no tenemos suficiente información sobre cierto aspecto o desconocemos el comportamiento preciso del sistema.
2. No queremos especificar un aspecto del sistema porque no es suficientemente interesante o importante y por lo tanto no queremos invertir mucho tiempo en su especificación.
3. Queremos especificar un sistema tan general como sea posible.

Al establecer la condición $0 < depRepLim$ estamos subespecificando el sistema porque queremos especificar un sistema lo más general posible. El sistema se va a comportar de la misma forma sin importar el valor específico de *depRepLim*; es decir, el sistema va a informar todos los depósitos cuyo monto sea de *depRepLim* pesos o más aun si no damos un valor específico para *depRepLim*. De esta forma nuestra especificación es un modelo correcto para cualquier valor de *depRepLim*.

Veamos un ejemplo del segundo motivo para subespecificar. Consideremos un sistema que debe computar la suma de verificación (*checksum*) de una secuencia de bytes. Como por el

momento no queremos dar los detalles de este algoritmo lo vamos a subespecificar. Entonces definimos una máquina configurada con dos parámetros: *BYTE*, que representa el tipo de los bytes (por lo que lo escribimos en mayúscula); y *checksum*, que representa el algoritmo que calcula la suma de verificación. Formalmente, tenemos lo siguiente:

MACHINE *CommSystem(BYTE, checksum)*
CONSTRAINTS $checksum \in \text{seq}(BYTE) \to BYTE$
. . . la descripción de la máquina de estados continúa aquí. . .
END

Es decir, la máquina *CommSystem* funciona correctamente solo si *checksum* es una función total que toma una secuencia de *BYTE* y retorna un *BYTE*. Eso es todo. Implícitamente *checksum* calcula la suma de verificación de cualquier secuencia de bytes. Al menos por el momento no estamos interesados en especificar los detalles del algoritmo que calcula la suma de verificación. Aun así podemos usar *checksum* en la definición de la máquina:

MACHINE *CommSystem(BYTE, checksum)*
CONSTRAINTS $checksum \in \text{seq}(BYTE) \to BYTE$
. . . .
OPERATIONS
. . . .
add$(msg, chk) \mathrel{\widehat{=}}$
PRE $msg \in \text{seq}(BYTE) \wedge chk \in BYTE \wedge checksum(msg) = chk$
THEN $msgs := msgs \cup \{msg\}$
END;
. . . .
END

5. Introducción a la verificación de máquinas de estado

¿Es correcta la especificación? ¿Cómo podemos saber si la especificación es o no correcta? Las especificaciones se escriben más que nada por dos motivos: para entender el problema antes de programar una implementación; y para que los programadores escriban la implementación desde la especificación (en lugar de hacerlo desde los requerimientos). Pero, ¿qué pasa si la especificación no es correcta? ¿Es posible? Sí, la especificación puede tener errores. Si entregamos una especificación incorrecta a los programadores es más que probable que la implementación reproduzca los errores de la especificación.

Como ya mencionamos, la especificación se escribe a partir de los requerimientos los cuales son una descripción informal de la funcionalidad del sistema. En consecuencia, es posible que durante el proceso de formalizar los requerimientos el ingeniero pueda cometer errores o pueda malinterpretar los requerimientos lo que lleva a tener una especificación incorrecta. De hecho, uno de los motivos para escribir una especificación es entender los requerimientos.

Si bien una especificación puede tener errores, normalmente, va a tener muchos menos errores que la implementación. Al mismo tiempo esos errores son mucho más simples de

encontrar que los errores que aparecen en la implementación. Además, detectar errores en la especificación es menos costoso que detectar los mismos errores durante el testeo o cuando el programa se pone en producción.

Las dos formas más usuales de encontrar errores en una especificación son:

1. Ejecutar algo parecido a casos de prueba, llamadas *simulaciones*.
2. Demostrar matemáticamente que la especificación está libre de ciertas clases de errores. En otras palabras, *demostrar matemáticamente* que la especificación verifica ciertas propiedades.

   Si logramos demostrar que la especificación verifica ciertas propiedades que surgen de manera obvia a partir de los requerimientos, entonces vamos a tener más confianza en que la especificación es una formalización fidedigna de los requerimientos. En muchos casos la especificación de las propiedades es mucho más simple que la especificación del sistema.

La primera técnica (simulaciones) la vamos a introducir cuando estudiemos $\{log\}$ hacia el fin del cuatrimestre. Por el momento vamos a presentar de manera simplificada la segunda técnica (demostraciones). En general todos los lenguajes formales de especificación están preparados para realizar demostraciones matemáticas sobre sus especificaciones. En el Método B esta técnica se denomina *verificación de máquinas de estado*. La verificación garantiza que la especificación no tendrá ciertas clases de errores. En esta sección presentaremos algunos aspectos de la verificación de máquinas de estados que luego veremos en funcionamiento cuando estudiemos $\{log\}$.

El núcleo de la verificación de máquinas de estado es *demostrar que cada operación preserva las invariantes de estado*. Las invariantes de estado son los predicados escritos en la sección Invariant. Conceptualmente una invariante de estado, o simplemente invariante, es un predicado que es verdadero en todos los estados de la máquina. Las invariantes de estado son importantes porque permiten capturar propiedades globales de las máquinas de estado.

Recuerden la invariante de estado de la máquina *Bank* (vean la Figura 3):

$$sa \in NIC \rightarrowtail \mathbb{N} \qquad (2)$$

Claramente esta propiedad surge de manera obvia de los requerimientos puesto que establece que:

- Cada caja de ahorros tiene un único saldo
- El saldo de cualquier caja de ahorros nunca es negativo

Ambas propiedades son importantes para el banco porque garantizan que ningún cliente va a ver dos saldos distintos de su cuenta al mismo tiempo (la primera), y que ningún cliente le va a deber dinero al banco (la segunda). En consecuencia es importante garantizar que nuestra especificación verifica estas propiedades. Si logramos *demostrar* que estas propiedades valen en todo estado del banco entonces estaremos más seguros sobre la consistencia de la especificación. Por el contrario, si entregáramos a los programadores una especificación que no verifica alguna de estas propiedades, es muy probable que terminemos poniendo un producción un sistema con fallas graves. Está claro que cuantas más propiedades logremos verificar sobre la especificación menores serán las posibilidades de tener una especificación que no es una formalización fidedigna de los requerimientos.

Para demostrar que una invariante es verdadero en todos los estados de una máquina tenemos que demostrar dos cosas:

1. El estado inicial *satisface* la invariante.
2. Cada operación *preserva* la invariante. Una operación preserva una invariante si toda vez que la operación inicia su ejecución en un estado que satisface la invariante la termina en un estado que también lo satisface.

Antes de presentar las reglas de la verificación de máquinas de estado necesitamos refrescar el siguiente concepto.

Sustitución

Sea P un predicado que depende de la variable v. Sea e una expresión del mismo tipo de v. Escribimos $P[v \mapsto e]$ para denotar la sustitución de v por e en P. Es decir, $P[v \mapsto e]$ es igual a P excepto que v ha sido sustituida por e.

Los siguientes son ejemplos de sustituciones:

- $sa \in NIC \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{N}[sa \mapsto \{\}]$ el cual reduce a $\{\} \in NIC \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{N}$
- $sa \in NIC \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{N}[sa \mapsto sa \Lleftarrow \{an \mapsto sa(an) + amt\}]$ el cual reduce a $sa \Lleftarrow \{an \mapsto sa(an) + amt\} \in NIC \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{N}$

Ahora presentamos las reglas de la verificación de máquinas de estado. Cada regla se denomina *condición de verificación*[5] u *obligación de prueba*[6]. Cuando demostramos una de estas condiciones de verificación decimos que la hemos *descargado*[7]. Supongamos que tenemos la siguiente máquina:

MACHINE M
VARIABLES v
INVARIANT Inv
INITIALISATION $v := Init$
OPERATIONS
  $y \leftarrow \mathbf{op}(x) \mathrel{\widehat{=}}$ **PRE** Pre **THEN** $v := Post \parallel y := Out$ **END**;
. . . .
END

Entonces consideramos las siguientes condiciones de verificación:

1. *Lema de inicialización*. El estado inicial satisface la invariante:

$$Inv[v \mapsto Init]$$

2. *Prueba de satisfacibilidad*. Toda transición de estado es satisfacible y puede modificar el estado:

$$\exists v, x, y.(Pre \land v \neq Post \land y = Out)$$

[5]'verification condition', en inglés.
[6]'proof obligation', en inglés.
[7]'discharged', en inglés.

3. *Prueba de satisfacibilidad*. Toda consulta de estado es satisfacible:

$$\exists v, x, y.(Pre \wedge y = Out)$$

4. *Lema de invarianza*. Toda transición de estado preserva la invariante:

$$\forall x.(Inv \wedge Pre \Rightarrow Inv[v \mapsto Post])$$

> Un lema de invarianza establece que si la invariante es verdadero en algún estado que a su vez satisface la precondición de una operación, entonces la invariante debe satisfacerse en el estado de llegada definido por la operación.

Es decir, si la operación parte de un estado que satisface la invariante, llega a un estado que también lo satisface. Por esta razón decimos que la operación *preserva* la invariante.

$Inv[v \mapsto Post]$ se puede leer como "la invariante evaluada en el estado de llegada".

Las consultas de estado preservan la invariante trivialmente (puesto que no modifican el estado).

Las condiciones de verificación más importantes y más difíciles de demostrar son los lemas de invarianza. El lema de inicialización y las pruebas de satisfacibilidad son necesarias, en parte, para evitar que un lema de invarianza se demuestre trivialmente porque el antecedente es insatisfacible (o sea el antecedente es una contradicción).

Veamos algunos ejemplos de condiciones de verificación correspondientes a la especificación de la Figura 3.

- El estado inicial satisface la invariante.

$$\begin{aligned} & sa \in NIC \nrightarrow \mathbb{N}[sa \mapsto \{\}] && \text{[by def. sustitución]} \\ & \Leftrightarrow \{\} \in NIC \nrightarrow \mathbb{N} && \text{[by conjunto vacío es función parcial]} \\ & \Rightarrow true \end{aligned}$$

  Obligación de prueba descargada.

- **deposit** es satisfacible y puede modificar el estado. Al aplicar (1) obtenemos:

$$\begin{aligned} & \exists sa, an, amt, msg. \\ & \quad (an \in NIC \wedge amt \in \mathbb{N} \\ & \quad \wedge (an \in \mathrm{dom}(sa) \wedge 0 < amt \Rightarrow sa \neq sa \oplus \{an \mapsto sa(an) + amt\} \wedge msg = ok) \\ & \quad \wedge (an \notin \mathrm{dom}(sa) \Rightarrow msg = nicNotExists) \\ & \quad \wedge (amt = 0 \Rightarrow msg = amountError)) \end{aligned}$$

  Esta fórmula se satisface con $sa = \{an \mapsto 0\} \wedge amt = 1 \wedge msg = ok$.

  Obligación de prueba descargada.

- **checkBalance** es satisfacible.

$$\begin{aligned}&\exists sa, an, bal, msg.\\&\quad(an \in NIC\\&\quad\land (an \in \operatorname{dom}(sa) \Rightarrow bal = sa(an) \land msg = ok)\\&\quad\land (an \notin \operatorname{dom}(sa) \Rightarrow msg = nicNotExists))\end{aligned}$$

  Esta fórmula se satisface con $sa = \{an \mapsto 0\} \land bal = 0 \land msg = ok$.
  Obligación de prueba descargada.

- **withdraw** preserva la invariante. Consideramos solo el caso cuando se modifica *sa* (el caso más complejo):

$$\begin{aligned}&\forall an, amt.\\&\quad(sa \in NIC \nrightarrow \mathbb{N} && \text{[invariante]}\\&\quad\land an \in NIC \land amt \in \mathbb{N} \land n \in \operatorname{dom}(sa) \land 0 < amt \land amt \leq sa(an) && \text{[precondición]}\\&\quad\Rightarrow sa \oplus \{an \mapsto sa(an) - amt\} \in sa \in NIC \nrightarrow \mathbb{N}) && \text{[invariante en el estado de llegada]}\end{aligned}$$

  Usando eliminación podemos ignorar la cuantificación universal, obteniendo:

$$\begin{aligned}&sa \in NIC \nrightarrow \mathbb{N}\\&\land an \in NIC \land amt \in \mathbb{N} \land n \in \operatorname{dom}(sa) \land 0 < amt \land amt \leq sa(an)\\&\Rightarrow sa \oplus \{an \mapsto sa(an) - amt\} \in sa \in NIC \nrightarrow \mathbb{N}\end{aligned}$$

  La demostración queda como Ejercicio 38.

Recuerden que todas estas condiciones de verificación se deben demostrar para *todas* las operaciones.

Como ya hemos mencionado, la asignación abstracta esconde un predicado. El lema de invarianza muestra cuál es el predicado que se oculta detrás de una asignación abstracta. Más precisamente, si P es un predicado que depende de la variable v y queremos saber si P es o no verdadero luego de ejecutar $v := expr$, entonces todo lo que tenemos que hacer es evaluar $P[v \mapsto expr]$.

Recuerden que siempre que sea posible debemos evitar usar cuantificadores en las especificaciones. Una de las razones para hacer esto es que demostrar propiedades que involucran cuantificadores es, en general, más difícil que probar propiedades libres de cuantificadores. Por otro lado, no es inusual que una especificación con cuantificadores sea más fácil de escribir y entender que una sin cuantificadores. Cuando escribimos especificaciones la cuestión más importante es escribir fórmulas que sean *obviamente* correctas. La especificación formal trata siempre sobre hacer que algo sea obviamente correcto porque los programas casi nunca lo son. Por lo tanto, podemos empezar escribiendo una especificación con cuantificadores que sea obviamente correcta para luego reescribirla sin cuantificadores. Incluso podríamos demostrar que la reescritura genera una especificación equivalente a la original de forma tal de estar seguros que podemos sustituir una especificación por la otra.

La discusión sobre evitar especificaciones formales que incluyen cuantificadores tiene sentido más que nada cuando se prevé poner en práctica alguna forma de *verificación asistida por computadora*. Es decir, evitar los cuantificadores en una especificación es importante cuando las

condiciones de verificación van a ser descargadas usando alguna herramienta con capacidad de efectuar demostraciones automáticas. Debido a consideraciones teóricas estas herramientas son más efectivas y eficientes cuando las demostraciones incluyen la menor cantidad posible de predicados cuantificados. Más adelante vamos a usar {*log*} para descargar automáticamente diversas condiciones de verificación.

En cualquier caso el mero hecho de escribir una especificación formal, como las que escribimos en B, es suficiente para detectar la mayoría de los errores, ambigüedades e incompletitudes en los requerimientos. La demostración de propiedades que surgen de una especificación representa un valor agregado que en general produce ganancias marginales comparadas con las que se obtienen al escribir la especificación misma.

6. Tipos y conjuntos

En este curso vamos a distinguir entre tipos y conjuntos. *Todos los tipos son conjuntos, pero no todos los conjuntos son tipos*. Los siguientes conjuntos son tipos.

- Cualquier conjunto declarado en la sección SETS es un tipo.
- Cualquier conjunto declarado como un parámetro de configuración de una máquina es un tipo.
- El conjunto $\mathbb{Z}$ es un tipo.
- Si X e Y son tipos luego el conjunto $X \times Y$ es un tipo.
  Los elementos de este tipo son pares ordenados.
- Si X es un tipo luego del conjunto $\mathbb{P} X$ es un tipo.
  Los elementos de este tipo son conjuntos.

Los conjuntos que no pueden construirse como se describe arriba no son tipos. Siguiendo esas reglas podemos concluir lo siguiente:

- $\mathbb{N}$ no es un tipo.
- Si X e Y son tipos, luego $X \leftrightarrow Y$ es un tipo (porque $X \leftrightarrow Y = \mathbb{P}(X \times Y)$).
- Si X e Y son tipos, luego $X \nrightarrow Y$ no es un tipo.
- Si X e Y son tipos, luego $X \rightarrow Y$ no es un tipo.
- Si X es un tipo, luego $\operatorname{seq}(X)$ no es un tipo.

Los conjuntos que son tipos lo son porque son *cerrados* bajo ciertas operaciones. Por ejemplo, $\mathbb{Z}$ es un tipo porque si $n, m \in \mathbb{Z}$ entonces tenemos que $x + y, x * y, x - y, x \operatorname{div} y, x \operatorname{mod} y \in \mathbb{Z}$. Por lo tanto, $\mathbb{Z}$ es cerrado bajo $+, *, -, \operatorname{div}, \operatorname{mod}$. Por lo contrario, $\mathbb{N}$ no es un tipo porque si $n, m \in \mathbb{N}$ entonces $x - y \in \mathbb{N}$ no es necesariamente verdadero. En consecuencia, $\mathbb{N}$ no es cerrado bajo la resta. Entonces no puede ser un tipo (a menos que eliminemos la resta de las operaciones).

Una declaración de variable de la forma:

$$n \in \mathbb{N}$$

se dice que no está *normalizada* porque $\mathbb{N}$ no es un tipo sino un conjunto. La declaración se puede *normalizar* al declarar $n \in \mathbb{Z}$ y conjugar $n \in \mathbb{N}$ o $0 \leq x$ en la sección INVARIANT. Por ejemplo, si tenemos:

MACHINE M
....
VARIABLES v
INVARIANT $v \in \mathbb{N} \wedge \dots$
....
END

podemos normalizar la máquina escribiendo lo siguiente:

MACHINE M
....
VARIABLES v
INVARIANT $v \in \mathbb{Z} \wedge 0 \leq v \wedge \dots$
....
END

Por consiguiente, una declaración de la forma $x \in A$ no está normalizada si A no es un tipo sino conjunto. La declaración normalizada de x se obtiene sustituyendo A por su *tipo contenedor* y conjugando $x \in A$ en la sección INVARIANT. A continuación damos el tipo contenedor de algunos conjuntos que se usan frecuentemente en las especificaciones. Siempre que sea posible hay que evitar la introducción de conjuntos infinitos que no sean tipos.

CONJUNTO	TIPO CONTENEDOR
$\mathbb{N}, \mathbb{N}_1, m\,..\,n$	$\mathbb{Z}$
$X \rightarrow\!\!\!\!+ Y, X \rightarrow Y$	$X \leftrightarrow Y$
$\mathrm{seq}\, X$	$\mathbb{Z} \leftrightarrow X$

Luego, la declaración de *sa* en nuestra especificación de las cajas de ahorros no está normalizada:

MACHINE *Bank*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \rightarrow\!\!\!\!+ \mathbb{N}$
....
END

La podemos normalizar como sigue:

MACHINE *Bank1*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \leftrightarrow \mathbb{Z} \wedge sa \in NIC \rightarrow\!\!\!\!+ \mathbb{N}$
....
END

aunque aun tenemos $\mathbb{N}$ en la invariante. Lo podemos eliminar como sigue:

MACHINE *Bank2*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \leftrightarrow \mathbb{Z} \land sa \in NIC \rightarrowtail \mathbb{Z} \land \forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$
....
END

Sin embargo tenemos restricciones solapadas: $sa \in NIC \leftrightarrow \mathbb{Z} \land sa \in NIC \rightarrowtail \mathbb{Z}$. En efecto, ambas restricciones establecen que el dominio de *sa* es *NIC* y el rango es $\mathbb{Z}$. Asumamos que existe un predicado, llamado *pfun*, que es verdadero solo si su argumento es una función parcial. Por lo tanto, podemos reescribir la invariante como sigue:

MACHINE *Bank3*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \leftrightarrow \mathbb{Z} \land pfun(sa) \land \forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$
....
END

De esta forma descompusimos la invariante original en una invariante de tipo ($sa \in NIC \leftrightarrow \mathbb{Z}$) y dos restricciones ($pfun(sa)$ y $\forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$) que no pueden ser verificadas con un *typechecker*.

Notar que $\forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$ se podría escribir como $\mathrm{ran}(sa) \subseteq \mathbb{N}$. Preferimos la primera porque evita la introducción de un conjunto infinito ($\mathbb{N}$). Por otro lado, más de una vez hemos dicho que hay que evitar la introducción de cuantificadores universales. Sin embargo, en este caso, la fórmula cuantificada es de una forma muy particular que se llama *cuantificador restringido*. Un cuantificador restringido es una fórmula de la forma $\forall x.(x \in A \Rightarrow p)$ donde A es un conjunto, llamado *dominio*, y p es una conjunción de predicados atómicos. En particular en el cuantificador restringido que hemos introducido el dominio es un conjunto finito. Cuando el dominio es finito los cuantificadores restringidos se pueden resolver automáticamente en muchos casos.

Normalización y condiciones de verificación. Normalizar una declaración puede tener impacto cuando se descargan obligaciones de prueba. En general, las declaraciones no-normalizadas tienden a ser más fáciles de leer y entender pero tienden a ser más complejas de descargar automáticamente. Por ejemplo, en {*log*} $x \in \mathbb{N}$ no se puede escribir pero $0 \leq x$ es en general equivalente y mucho más simple de procesar de forma automática. Lo mismo ocurre con *Bank*1, *Bank*2 y *Bank*3. *Bank*1 no se puede escribir en {*log*} aunque las otras dos sí se pueden escribir y {*log*} tenderá a descargarlas automáticamente.

7. Designaciones

Lo que sigue es la especificación de una de las operaciones definidas sobre una caja de ahorros.

$my0 \longleftarrow \textbf{MQ1xb}(ww, x23) \mathrel{\widehat{=}}$
PRE $ww \in N \land x23 \in \mathbb{N}$
THEN
SELECT $ww \in \text{dom}(yy) \land 0 < x23 \land x23 \leq yy(ww)$
THEN $yy(ww), my0 := yy(ww) - x23, e0$
WHEN $ww \notin \text{dom}(yy)$ **THEN** $my0 := e1$
WHEN $x23 = 0$ **THEN** $my0 := e2$
WHEN $ww \in \text{dom}(yy) \land yy(ww) < x23$ **THEN** $my0 := e3$
END
END;

¿Puede adivinar de cuál se trata? ¿Es razonable que tenga que adivinar? ¿Qué hace que la especificación anterior sea mejor que esta? En esta versión, ¿está mal la lógica? Un programador que desconoce los requerimientos específicos del banco, ¿puede saber qué está programando? Imagine que el programador se las arregla para programarla (¿es posible, podría hacerlo?), ¿con cuál opción de menú hay que asociar la ejecución del programa resultante? ¿Hay alguna forma de saberlo?

La especificación corresponde a la operación **withdraw**. Es el resultado de sustituir los identificadores usados en la especificación original por otros menos descriptivos. La lógica no está mal, sólo es difícil leerla y asociarla con el requerimiento del usuario. Toda la confusión surge de la elección de identificadores que hizo el ingeniero, pero, ¿qué obligación tiene de elegir unos nombres sobre los otros? ¿Es una restricción de B? Es decir, ¿cambia la semántica de una especificación B al elegir un conjunto de identificadores en lugar de otro? La semántica de una especificación B está dada por el lenguaje en sí y no por su relación con los requerimientos.

Los identificadores usados en la especificación viven en el mundo de la lógica, por eso forman un modelo (que es una abstracción de la realidad). Las necesidades de una comunidad de usuarios son reales, no *son* parte de la lógica ni *son* un modelo. Un modelo es una abstracción de la realidad, por lo tanto no *es* la realidad. Sin embargo, hacemos el modelo para luego escribir un programa que finalmente es usado en el mundo real. En consecuencia hay una relación entre el modelo y la realidad. Si esa relación se pierde, como ocurrió con la operación **MQ1xb**, se podrá escribir un programa que cumpla con lo estipulado en esa operación pero este programa nunca podrá ser usado correctamente en la realidad: sería una casualidad que el programa correspondiente a **MQ1xb** se asocie con la opción de menú correcta pues no hay en el esquema ninguna señal que indique el requerimiento que está modelando.

Por lo tanto, resulta esencial preservar la relación entre el modelo y la realidad. Supongamos que antecedemos la definición de la máquina que incluye a **MQ1xb** con lo siguiente:

a es un número de caja de ahorros $\approx a \in N$

El saldo de la caja de ahorros $a \approx yy(a)$

La operación ejecutó correctamente $\approx e0$

El número de caja de ahorros no existe $\approx e1$

El monto a extraer es incorrecto $\approx e2$

La caja de ahorros no tiene fondos suficientes $\approx e3$

Se extrae el monto m de la caja de ahorros a comunicando el resultado r de la operación al usuario $\approx r \longleftarrow MQ1xb(a, m)$

¿Podría ligar ahora el modelo con la realidad? ¿Sabe qué requerimiento está modelando la operación **MQ1xb**? ¿Puede asociarle la opción de menú correcta?

Claramente ahora podría hacerlo. Lo que acabamos de hacer es *designar* los términos formales primitivos del modelo. Cada designación liga un término formal del modelo con un *fenómeno* de interés en el mundo real. Los fenómenos de interés son eventos, operaciones o acciones y constantes. La relación entre el término formal y el mundo real es necesariamente informal porque el mundo real lo es. El símbolo $\approx$ demarca la frontera entre el mundo real (a su izquierda) y el mundo formal o lógico (a la derecha). Sin esta documentación el modelo es nada más que una teoría axiomática más sin conexión con la realidad, es el trabajo de un matemático y no de un informático.

Por lo tanto, toda especificación debe estar precedida por la designación de todos los términos formales. En B estos comprenden únicamente:

- Los tipos básicos (*NIC*). En este caso las designaciones son de la forma:
  - n es un número de cuenta $\approx NIC(n)$

  o de la forma:
  - a es un auto $\approx a \in AUTO$
- Los nombres de las operaciones con sus parámetros de entrada y de salida (**withdraw**)
- Los nombres de las variables de estado (*sa*)
- Los nombres de los parámetros de la máquina (*depRepLim*)
- Los nombres de los elementos de conjuntos dados por extensión en la cláusula Sets (*ok*)

La existencia de las designaciones no prohíbe ni desalienta el uso de nombres sugerentes para los términos designados. Una de las grandes ventajas de contar con un modelo formal de los requerimientos es que se posee un documento claro, sin ambigüedades, legible, etc. Sería una tontería empañar este logro con nombres extraños.

Notar que la designación no describe ningún requerimiento. Se trata de dejar bien claro el vocabulario esencial del dominio de aplicación del que tratan los requerimientos. De la misma forma, la cantidad de designaciones tiene que ser la mínima posible. Por ejemplo, el estado inicial de una caja de ahorros no se designa, sino que se define (como mencionamos más arriba).

Tener presente que las designaciones se deben documentar antes de modelar y no luego. En este apunte se procedió a la inversa para poner énfasis en B y no comenzar con algo que es ajeno al lenguaje y propio de una metodología de desarrollo que puede aplicarse con cualquier lenguaje de especificación formal.

8. Cada titular puede tener más de una caja de ahorros y viceversa

En esta sección modelaremos un sistema de cajas de ahorros un poco más real. En este sistema una caja de ahorros puede estar asociada a varios titulares (clientes), y cada titular puede tener varias cajas de ahorros. Cualquier titular puede extraer, depositar y consultar el

saldo de sus cajas. Cualquier persona puede hacer un depósito en cualquier cuenta pero solo los titulares pueden hacer extracciones de las suyas. Además, modelaremos la operación de transferencia de dinero entre cajas de ahorro. Otro requerimiento importante es que el banco desea llevar un registro de los depósitos y extracciones de cada cuenta (esto se llama historia de la cuenta). Finalmente, el banco desea registrar el DNI, nombre completo y domicilio de cada cliente.

8.1. Tipos elementales y variables de estado

Nuevamente comenzamos seleccionando los tipos elementales para nuestro modelo. Necesitamos registrar el DNI, nombre completo y domicilio de cada cliente. Sólo registrarlos; aparentemente no hay que realizar ninguna operación compleja con estos datos. Por lo tanto, no nos interesa la estructura de los nombres ni de los domicilios. Por otro lado, el DNI sirve para identificar a una persona. Entonces podemos agrupar nombre completo y domicilio en un tipo básico y definir un tipo básico para los DNI, de la siguiente forma.

MACHINE *Bank*
SETS *NIC*; *DNI*; *PDATA*
    . . . la descripción de la máquina de estados continúa aquí. . .
END

Una de las decisiones fundamentales, dados los nuevos requerimientos, es determinar cómo se define el estado del sistema, lo que implica elegir la forma de relacionar entre sí:

- Los DNI de los clientes
- Los datos personales de los clientes
- Los números de cuenta
- El saldo de cada cuenta
- La historia de cada cuenta.

  Para registrar la historia de una cuenta vamos a usar una secuencia de enteros porque: (a) es necesario preservar el orden en que se hicieron los depósitos y las extracciones; y (b) los depósitos se registrarán como números positivos y las extracciones como números negativos.
- Los titulares de cada caja de ahorros

Con requisitos de esta complejidad no hay una única solución ni existe la mejor solución. Varias soluciones posibles tienen ventajas y desventajas. Algunas más que otras. Varias de ellas son más o menos equivalentes. Es conveniente, pues, elegir una solución entre ese grupo de las mejores. Cualquier solución debe tener en cuenta al menos los siguientes criterios:

- Se debe poder expresar todos los requerimientos.
- La especificación debe ser lo más evidentemente correcta que sea posible.
- El modelo debe ser suficientemente abstracto como para no invalidar tempranamente implementaciones razonables.

Dados estos requisitos, nuestra elección para modelar el estado del sistema de cajas de ahorros son las siguientes variables:

- $ab \in NIC \pfun \mathbb{N}$, por *account balance*, que asocia cada *NIC* a su saldo.
- $pd \in DNI \pfun PDATA$, por *personal data*, que asocia cada *DNI* a los datos personales de ese individuo.
- $ah \in NIC \pfun \operatorname{seq} \mathbb{Z}$, por *account history*, que asocia cada *NIC* con la historia de esa cuenta.
- $ao \in NIC \leftrightarrow DNI$, por *account owner*, que asocia cada *NIC* con el *DNI* de uno o más titulares. Observar que en este caso elegimos una relación binaria (y no una función parcial) puesto que una persona puede tener varias cuentas y una cuenta puede tener varios titulares.

En consecuencia tenemos lo siguiente:

MACHINE *Bank*
SETS *NIC*; *DNI*; *PDATA*
VARIABLES ab, pd, ah, ao
. . . la descripción de la máquina de estados continúa aquí. . .
END

Las variables de estado, a su vez, están relacionadas entre sí. Por ejemplo, si en el banco existe la cuenta con número m entonces tanto ab como ah tienen que estar definidas en m. Más aun, m tiene que estar asociado a algún *DNI* en ao; y, a su vez, pd tiene que estar definida en ese *DNI*. Todas esas relaciones, más el tipo de cada variable, las vamos a codificar como invariantes puesto que se tienen que preservar siempre.

MACHINE *Bank*
SETS *NIC*
VARIABLES sa
INVARIANT $ab \in NIC \pfun \mathbb{N} \land pd \in DNI \pfun PDATA \land ah \in NIC \pfun \operatorname{seq} \mathbb{Z} \land ao \in NIC \leftrightarrow DNI$
$\land \operatorname{dom}(ab) = \operatorname{dom}(ah) \land \operatorname{dom}(ab) = \operatorname{dom}(ao) \land \operatorname{dom}(pd) = \operatorname{ran}(ao)$
. . . la descripción de la máquina de estados continúa aquí. . .
END

Por otra parte, hay una invariante muy importante que relaciona el saldo de cada cuenta con su historia: la suma de la historia tiene que ser igual al saldo de la cuenta. Para codificar esta invariante usamos el operador B llamado *suma generalizada*, de la siguiente forma.

MACHINE *Bank*
SETS *NIC*
VARIABLES sa
INVARIANT $ab \in NIC \pfun \mathbb{N} \land pd \in DNI \pfun PDATA \land ah \in NIC \pfun \operatorname{seq} \mathbb{Z} \land ao \in NIC \leftrightarrow DNI$
$\land \operatorname{dom}(ab) = \operatorname{dom}(ah) \land \operatorname{dom}(ab) = \operatorname{dom}(ao) \land \operatorname{dom}(pd) = \operatorname{ran}(ao)$
$\land \forall n.(n \in \operatorname{dom}(ab) \Rightarrow \sum i.(i \in \operatorname{dom}(ah(n)) \mid ah(n)(i) = ab(n))$
. . . la descripción de la máquina de estados continúa aquí. . .
END

La forma general de la suma generalizada es:

$\sum x.(P(x) \mid E(x))$

lo que representa la suma de todos los $E(x)$ para los cuales $P(x)$ se satisface.

En nuestro caso, combinamos la suma generalizada con el hecho de que las secuencias son funciones. De esta forma si $n \in \mathrm{dom}(ab)$ sabemos por una de las invariantes que $n \in \mathrm{dom}(ah)$ por lo que $ah(n)$ es una secuencia de números enteros. Si $ah(n)$ es una secuencia de números enteros entonces $ah(n)(i)$ es uno de los números de la secuencia siempre y cuando $i \in \mathrm{dom}(ah(n))$.

8.2. Las operaciones

En esta sección vamos a dar la especificación de dos operaciones para los nuevos requerimientos. En la siguiente sección damos la especificación de la operación para transferir dinero de una cuenta a otra. Las restantes operaciones quedan como ejercicio.

En la Figura 4 pueden ver la especificación de las operaciones **open** y **withdraw**. Si bien lucen más complejas que las que hemos visto hasta el momento se debe más que nada a que la máquina tiene más variables y hay que tener en cuenta más precondiciones o postcondiciones. La matemática que usamos es la misma que la usada en la especificación anterior excepto en la asignación a *ah* donde usamos el operador de secuencias que representa la concatenación ($\frown$). Notar que en esa postcondición escribimos $ah(an) \frown [-amt]$ lo que indica que en la historia de la caja de ahorros *an* la extracción se registra con $-amt$.

Un detalle que conviene resaltar es la precondición funcional $an \mapsto dni \in ao$ en **withdraw**, que indica que la persona con DNI *dni* es uno de los titulares de la caja de ahorros *an*. Seguramente se están preguntando por qué no escribimos, por ejemplo, $an \in \mathrm{dom}(ab)$ como hemos hecho en la especificación anterior. La justificación es que $an \mapsto dni \in ao$ más las invariantes implican $an \in \mathrm{dom}(ab)$. En efecto si $an \mapsto dni \in ao$ se satisface entonces $an \in \mathrm{dom}(ao)$ lo que a su vez, por la invariante $\mathrm{dom}(ab) = \mathrm{dom}(ao)$, implica $an \in \mathrm{dom}(ab)$. En consecuencia es suficiente con pedir $an \mapsto dni \in ao$.

8.3. Transferencia de dinero entre dos cuentas – Composición secuencial

Dejamos para una sección aparte la especificación de la operación que permite transferir dinero de una cuenta a otra porque tiene una particularidad interesante.

Consideren que la transferencia de *amt* pesos desde la cuenta *ian* a la cuenta *dan* se puede pensar como la extracción de *amt* pesos de la cuenta *ian* y el posterior depósito de la misma cantidad de dinero en la cuenta *dan*. En la Figura 5 pueden ver una representación gráfica de lo que acabamos de decir. La idea es que **transfer** llegue al mismo estado al que se arriba si primero ejecutamos **withdraw** y luego **deposit**.

En B este tipo de especificaciones se logran usando la *composición secuencial* de sentencias. En nuestro caso las sentencias B que queremos componer secuencialmente son las operaciones **withdraw** y **deposit**. Si S y T son dos sentencias B, la composición secuencial de S con T se escribe $S;\ T$. Entonces, usando composición secuencial la especificación de **transfer** es la siguiente:

$msgw, msgd \leftarrow \mathbf{transfer}(dni, ian, dan, amt) \mathrel{\widehat{=}}$
$\quad msgw \leftarrow \mathbf{withdraw}(dni, ian, amt);\ msgd \leftarrow \mathbf{deposit}(dan, amt)$
$\quad \mathbf{END};$

MACHINE $Bank$
SETS $NIC;\ DNI;\ PDATA;$
$MSG = \{ok, nicExists, nicNotExists, amountError,$
$insufficientFunds, balanceNotZero, accessError\}$
VARIABLES ab, pd, ah, ao
INVARIANT $ab \in NIC \nrightarrow \mathbb{N} \wedge pd \in DNI \nrightarrow PDATA \wedge ah \in NIC \nrightarrow \mathrm{seq}\,\mathbb{Z} \wedge ao \in NIC \leftrightarrow DNI$
$\wedge\ \mathrm{dom}(ab) = \mathrm{dom}(ah) \wedge \mathrm{dom}(ab) = \mathrm{dom}(ao) \wedge \mathrm{dom}(pd) = \mathrm{ran}(ao)$
$\wedge\ \forall n.(n \in \mathrm{dom}(ab) \Rightarrow \sum i.(i \in \mathrm{dom}(ah(n)) \mid ah(n)(i)) = ab(n))$
INITIALISATION $ab := \{\} \parallel pd := \{\} \parallel ah := \{\} \parallel ao := \{\}$
OPERATIONS
$msg \longleftarrow$ **open**$(an, dni, data) \mathrel{\widehat{=}}$
PRE $an \in NIC \wedge dni \in DNI \wedge data \in PDATA$
THEN **IF** $an \notin \mathrm{dom}(ab)$
THEN $ab := ab \cup \{an \mapsto 0\}$
$\parallel pd := pd \cup \{dni \mapsto data\}$
$\parallel ah := ah \cup \{an \mapsto [\,]\}$
$\parallel ao := ao \cup \{an \mapsto dni\}$
$\parallel msg := ok$
ELSE $msg := nicExists$
END;

$msg \longleftarrow$ **withdraw**$(dni, an, amt) \mathrel{\widehat{=}}$
PRE $dni \in DNI \wedge an \in NIC \wedge amt \in \mathbb{N}$
THEN
SELECT $an \mapsto dni \in ao \wedge 0 < amt \wedge amt \leq sa(an)$
THEN $ab(an),\ ah(an),\ msg := ab(an) - amt,\ ah(an) \frown [-amt],\ ok$
WHEN $an \mapsto dni \notin ao$ **THEN** $msg := accessError$
WHEN $amt = 0$ **THEN** $msg := amountError$
WHEN $an \in \mathrm{dom}(sa) \wedge sa(an) < amt$ **THEN** $msg := insufficientFunds$
END
END
END

Figura 4: Especificación de algunas operaciones del sistema de cajas de ahorros más complejo

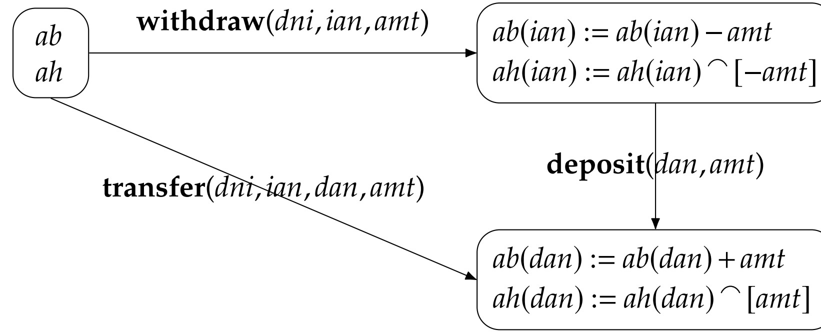


Figura 5: **transfer** como la composición secuencial de **withdraw** y **deposit**

Observen que tenemos que usar dos parámetros de salida (*msgw* y *msgd*) porque cada una de las operaciones que invocamos produce su propia salida. Si usáramos un único parámetro tendríamos una variable que podría tener dos valores diferentes en la misma fórmula lógica.

La semántica de $S; T$ es la siguiente. S se ejecuta primero, si no termina entonces $S; T$ no termina, y T nunca se ejecuta. Si la ejecución de S termina, entonces T se ejecuta desde el estado de llegada de S. Si T no termina tampoco lo hace $S; T$. El estado al cual arriba T es el estado de llegada de $S; T$. Es decir que S parte de un cierto estado y, si termina, llega a un estado desde el cual parte T quien a su vez, si termina, llega al estado final.

9. El sistema elije el número de cuenta — La sentencia ANY

En la Figura 3 podemos ver que al abrir una caja de ahorros la operación **open** espera que el invocador provea el número de cuenta. Esto no es lo usual porque se supone que el sistema debería seleccionar un número de cuenta nuevo. A continuación vamos especificar una versión de **open** donde el sistema, internamente, selecciona el número de cuenta para la nueva caja de ahorros:

$accn \leftarrow$ **open** $\widehat{=}$
  PRE $true$
  THEN ANY an **WHERE** $an \in NIC \wedge an \notin \mathrm{dom}(sa)$ **THEN** $sa, accn := sa \cup \{an \mapsto 0\}, an$ **END**
  END

En primer lugar noten que la operación no espera ningún parámetro de entrada y no emite un mensaje como salida sino el valor *accn* que será el número de cuenta seleccionado por el sistema dado que de otra forma el usuario no podría conocer su número de cuenta. Observen también que la precondición ahora es simplemente *true*. Sin embargo lo más importante es el uso de la nueva sentencia **ANY** que permite seleccionar un valor arbitrario (no determinado, o sea no-determinista) que cumpla la condición que se especifica luego de **WHERE**. El valor seleccionado luego se puede usar en la sentencia que se especifica luego de **THEN**. En nuestro caso usamos *an* para establecer el nuevo estado del banco y lo emitimos al entorno. Es decir, no sabemos el valor específico de *an* aunque sí sabemos que es un *NIC* que no está usado como número de cuenta en el banco.

Cuando la precondición de una operación es *true* la sentencia **PRE** se puede eliminar:

$accn \longleftarrow$ **open** $\widehat{=}$
ANY an **WHERE** $an \in NIC \wedge an \notin \mathrm{dom}(sa)$ **THEN** $sa, accn := sa \cup \{an \mapsto 0\}, an$ **END**

La forma general de **ANY** es la siguiente:

ANY x **WHERE** Q **THEN** T **END**

donde x tiene que ser una variable nueva; Q es un predicado que al menos debe proveer un tipo para x; y T, llamado *cuerpo* de la sentencia **ANY**, es una sentencia del lenguaje B. **ANY** también acepta una lista de variables:

ANY $x_1, \ldots, x_n$ **WHERE** Q **THEN** T **END**

teniendo en cuenta que deben ser todas nuevas y que Q debe dar un tipo para todas ellas.

Existe una forma simplificada de la sentencia **ANY** que se denomina *asignación no-determinista*:

$x :\in S$

la cual asigna un valor arbitrario de S a la variable x. La asignación no-determinista se define de la siguiente forma:

$x :\in S \;\widehat{=}\;$ **ANY** y **WHERE** $y \in S$ **THEN** $x := y$ **END**

La asignación no-determinista suele usarse en la sección Initialisation cuando no sabemos qué valor exacto asignar a alguna de las variables de estado o no queremos asignar un valor exacto.

Claramente la sentencia **ANY** introduce no-determinismo en la especificación que cuando sea transformada en una implementación deberá ser eliminado.

En el Capítulo 9 del libro de B van a encontrar más información sobre **ANY** y otras sentencias que introducen no-determinismo en la especificación.

10. Ejercicios

1. Considere la máquina de estados con variables de estado x e y dada en la Sección 1. Escriba las transiciones que conectan el segundo y el tercer estado, el tercero con el cuarto y el cuarto con el primero.
2. Explique por qué un conjunto de la forma $\{1 \mapsto 3\}$ es una relación binaria. ¿Cuál es su tipo? ¿Es una función parcial? Si lo es, ¿cuál es su tipo?
3. Escriba todos los elementos del tipo $\mathbb{P}\,STATUS$ (Sección 1).
4. Escriba una relación binaria que no sea una función parcial.
5. Escriba tres elementos de los siguientes tipos o conjuntos, que no pertenezcan a los otros tipos y conjuntos dados en la lista.
   a) $\mathbb{Z}$
   b) $\mathbb{Z} \times \mathbb{Z}$
   c) $\mathbb{P}\mathbb{Z}$
   d) $\mathbb{Z} \leftrightarrow \mathbb{Z}$
   e) $\mathbb{Z} \nrightarrow \mathbb{Z}$
   f) $\mathrm{seq}(\mathbb{Z})$
6. Asuma que T es un tipo elemental y $x \in T \wedge a \in \mathbb{P}\,T$. Determine cuáles de los siguiente está correctamente tipado. Justifique.
   a) $x = \emptyset$
   b) $a = \emptyset$
   c) $x \in a$
   d) $x \notin a$
   e) $x \subseteq a$
   f) $\{x\} \subseteq a$
   g) $x = a$
   h) $\{x, x\} = a$
   i) $\{x\} = a \cup \{x\}$
   j) $\{x\} = \{a\}$
   k) $\{\{x\}\} = \{a\}$
   l) Si U es otro tipo y $b \in \mathbb{P}\,U$, $a = b = \emptyset$
7. Demuestre que si T es un tipo entonces $A \in \mathbb{P}\,T \Leftrightarrow A \subseteq T$.
8. Demuestre que la unión de dos relaciones binarias es una relación binaria.
9. Demuestre que la unión de dos funciones parciales no necesariamente es una función parcial.
10. Demuestre que la intersección de dos funciones parciales es una función parcial.
11. Según la definición de $\mathrm{seq}(X)$ dada en la Sección 1, determine el conjunto equivalente a esta secuencia y el tipo de X.

$$[3, 54, 12, 3]$$

12. Si es posible calcule lo siguiente; si no lo es, justifique.
    a) $\mathrm{dom}\,\emptyset$
    b) $\mathrm{dom}\{1 \mapsto a\}$
    c) $\mathrm{dom}\{1 \mapsto a, 1 \mapsto b\}$
    d) $\{1 \mapsto a, 2 \mapsto b\}(2)$
    e) $\{1 \mapsto a, 2 \mapsto b\}(3)$
    f) $\{1 \mapsto a, 2 \mapsto b\}(a)$
13. ¿Por qué el siguiente conjunto no es una función parcial?

$$\{nic_1 \mapsto 100, nic_2 \mapsto 230, nic_1 \mapsto 529\}$$

14. ¿Por qué tenemos un problema si el banco llega a un estado donde $n \mapsto b \in sa \wedge b < 0$? Lo mismo con: $n \mapsto b_1, n \mapsto b_2 \in sa \wedge b_1 \neq b_2$.
15. Demuestre que si $sa \in NIC \nrightarrow \mathbb{N} \wedge n \in NIC$ y $n \notin \mathrm{dom}\, sa$, entonces $sa \cup \{n? \mapsto 0\} \in NIC \nrightarrow \mathbb{N}$.
16. Complete la definición del tipo *MSG* agregando todos los mensajes que inicialmente no fueron considerados.
17. Demuestre que si $x \notin \mathrm{dom}\, R$ entonces $R \oplus \{x \mapsto y\}$ es igual a $R \cup \{x \mapsto y\}$. Seguidamente demuestre que si $\mathrm{dom}\, S \cap \mathrm{dom}\, R = \emptyset$ entonces $R \oplus S = R \cup S$.
18. Demuestre que si $f : X \nrightarrow Y$; $x : X$; $y : Y$ entonces $f \oplus \{x \mapsto y\} : X \nrightarrow Y$.
19. Determine las condiciones que hacen que $R \oplus S$ sea una función.
20. Demuestre que si $R : X \leftrightarrow Y$ y $x \notin \mathrm{dom}\, R$, entonces $\{x\} ⩤ R = R$.
21. Demuestre que si $R : X \leftrightarrow Y$ y $A : \mathbb{P}\, X$ entonces $(A \lhd R) \cap (A ⩤ R) = \emptyset$.
22. Demuestre que si $R : X \leftrightarrow Y$ y $A : \mathbb{P}\, X$ entonces $R = (A \lhd R) \cup (A ⩤ R)$.
23. Reemplace el siguiente **IF-THEN-ELSE** con una sentencia **SELECT**.

$$\textbf{IF} \quad an \notin \mathrm{dom}(sa) \quad \textbf{THEN} \quad sa, msg := sa \cup \{an \mapsto 0\}, ok \quad \textbf{ELSE} \quad msg := nicExists \quad \textbf{END}$$

24. Explique por qué incluimos $an \in \mathrm{dom}(sa)$ como una precondición en la tercera rama de **close**.
25. La postcondición de **checkBalance** en la rama **ELSE** no establece un valor para *bal*. ¿Es esto necesario? Si lo es, ¿cuál sería ese valor?
26. Digamos que el banco requiere un depósito inicial de 20.000 pesos al momento de abrir la cuenta. Especifique ese requerimiento.
27. Especifique una operación que emita una lista con los saldos de un grupo de cuentas.
28. Especifique una operación que cierre más de una cuenta.
29. En la especificación de **deposit** desarrollada en la Sección 4, hay ramas donde no se definen valores para *ra* y *rn*. ¿Qué significa esto? ¿Cuál sería el comportamiento correcto en esas ramas?
30. En relación al ejercicio anterior, modifique la especificación de **deposit** como para que emita un conjunto de pares ordenados de la forma $rn \mapsto ra$. ¿Cuál es el valor del nuevo parámetro de salida en las ramas analizadas en el problema anterior?

31. Estudie las secciones CONSTANTS y PROPERTIES de una máquina B. Define una constante de tipo *NIC* cuyo significado informal sea "no es un número de cuenta". Reescriba la especificación de **deposit** de forma tal que emite esta constante cuando no se debe enviar un reporte al banco central. ¿Cuál debería ser el valor para *ra* en este caso?
32. En relación al ejercicio anterior, ¿podemos hacer que **deposit** emita -1 en *ra* cuando no se debe enviar un reporte al banco central? ¿Es necesario definir un valor para *rn* en este caso?
33. Continuando con el mismo problema, haga lo siguiente:
    a) Subespecifique una función que convierta *NIC* en strings.
    b) Subespecifique una función que convierta $\mathbb{Z}$ en strings.
    c) Subespecifique la cadena vacía.
    d) Redefina **deposit** como para que emita dos strings en lugar de un *NIC* y un $\mathbb{N}$.
    e) Redefina **deposit** como para que emita la cadena vacía en cualquier otro caso.
    f) Complete la especificación de **deposit** con los comportamientos anormales que faltan.
34. Rehaga el problema anterior de la siguiente forma:
    a) Declare el tipo conjunto *REPORT*.
    b) Declare *empty* como una constante de tipo *REPORT*.
    c) Subespecifique una función que convierta $NIC \times \mathbb{N}$ en *REPORT*.
    d) Redefina **deposit** como para que emita un *REPORT*.
35. Analice cuál es la mejor solución para emitir el reporte al banco central. ¿Cuál debería ser el criterio para determinar cuál es la mejor solución?
36. Asuma que el banco central prohíbe extracciones de más de 20 millones de pesos. Especifique ese requerimiento.
37. Compruebe que $sa = \{an \mapsto 0\} \land amt = 1 \land msg = ok$ satisface la siguiente condición de verificación.

$$
\begin{aligned}
&\exists sa, an, amt, msg.\\
&\quad (an \in NIC \land amt \in \mathbb{N}\\
&\quad \land (an \in \mathrm{dom}(sa) \land 0 < amt \Rightarrow sa \neq sa \mathbin{\lhd\!\!\!+} \{an \mapsto sa(an) + amt\} \land msg = ok)\\
&\quad \land (an \notin \mathrm{dom}(sa) \Rightarrow msg = nicNotExists)\\
&\quad \land (amt = 0 \Rightarrow msg = amountError))
\end{aligned}
$$

38. Descargue el siguiente lema de invarianza.

$$
\begin{aligned}
&sa \in NIC \mathbin{+\!\!\!\rightarrow} \mathbb{N}\\
&\land an \in NIC \land amt \in \mathbb{N} \land n \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an)\\
&\Rightarrow sa \mathbin{\lhd\!\!\!+} \{an \mapsto sa(an) - amt\} \in sa \in NIC \mathbin{+\!\!\!\rightarrow} \mathbb{N}
\end{aligned}
$$

39. Explique por qué $X \mathbin{+\!\!\!\rightarrow} Y$ no puede ser un tipo.
40. Demuestre lo siguiente:

$$(\forall x, y.(x \mapsto y \in sa \bullet 0 \leq y)) \Leftrightarrow (\exists k.(k \in \mathbb{Z} \land \mathrm{ran}(sa) \subseteq 0 \mapsto k))$$

41. Demuestre que **withdraw** preserva la invariante.
42. Escriba y demuestre todas las condiciones de verificación que faltan sobre la especificación del sistema de cajas de ahorros.
43. Escriba un programa que implemente **open**.
44. En relación al ejercicio anterior, ¿cuáles son las diferencias entre el programa y la especificación? ¿Cómo implementó *NIC*? ¿Puede *garantizar* que su implementación verifica la especificación? ¿Cómo lo podría garantizar?
45. Escriba las designaciones de los requerimientos enunciados en la Sección 2.
46. Escriba las designaciones de los requerimientos enunciados en la Sección 8.
47. Respecto a la versión del sistema de cajas de ahorros de la Sección 8, especifique las siguientes operaciones: depositar dinero en una caja de ahorros, consultar el saldo de una caja de ahorros, cerrar una caja de ahorros, agregar un titular a una caja de ahorros (solo lo puede hacer un titular), eliminar un titular de una caja de ahorros (solo lo puede hacer un titular y la cuenta no puede quedar sin titulares).
48. Respecto al problema anterior, ¿puede especificar una operación que muestre las transacciones de una caja de ahorros efectuadas entre dos fechas? Si puede, hágalo; si no, reescriba el modelo teniendo en cuenta este requerimiento.
49. Redefina la operación **open** de la Figura 4 de manera tal que sea el sistema el que genera el número de cuenta para la nueva caja de ahorros.