DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico N°1

Especificación de TADs

10 de abril de 2025 — Algoritmos y estructura de datos

Grupo Trelew

Integrante	LU	Correo electrónico
Apellido, Nombre1	001/01	email1@dominio.com
Groisman, Salvador	243/24	salvagroisman@gmail.com
Apellido, Nombre3	003/01	email3@dominio.com
Apellido, Nombre4	004/01	email4@dominio.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Parte I
Introducción:

Parte II
Jutificativo paso a paso de las decisiones tomadas:

1. Observadores

1.1. Definiciones:

1. obs: cantidadBloques(): $\mathbb{Z}$.
2. obs: cantidadUsuarios(): $\mathbb{Z}$.
3. obs: saldo(id: $\mathbb{Z}$): $\mathbb{Z}$.
4. obs: transaccion(id.bloque: Z, id.transacción: Z): Transacción.
5. obs: bloque(id: $\mathbb{Z}$): Bloque.

1.2. Razones para hacerlo de esta forma:

1. Esto me devuelve cuantos bloques tengo, originalmente usamos cantidadEmitida, pero tras estar especificando un tiempo, nos dimos cuenta que esto nos permite simplificar algunos observadores.
2. Esto lo uso para evitar definirme un tipo de datos usuario, lo cual le dará al programador libertad de definirlo como prefiera (por ejemplo struct o diccionario, entre otras).
3. En este caso, nuevamente para evitar definir el tipo de usuario, nos centramos simplemente en definir un observador que nos permita acceder al saldo que tiene un usuario específico en una instancia del programa.
4. Aquí nos limitamos a definir lo que pide el enunciado. Estamos suponiendo que el tipo Transacción es una tupla como la dada.
5. Aquí estamos pidiendo que dada la id de un bloque, el observador devuelva la lista de transacciones. Nuevamente aquí suponemos que el tipo Bloque es una secuencia de transacciones.

2. Inicializador

2.1. Definición:

`proc inicalizarBerretacoin` () :
`requiere` $\{\}$
`asegura` $\{res.cantidadBloques() = 0\}$
`asegura` $\{res.cantidadUsuarios() = 0\}$

2.2. Pequeño comentario:

Inicializamos el programa, la cantidad de usuarios es 0, pues nos interesa además que esta Id quede reservada para el emisor de la criptomoneda

3. Usuarios:

3.1. Definiciones de Procesos:

1. `proc agregarUsuario` (`inout` b: Berretacoin)
   `requiere` $\{b = b_0\}$
   `asegura` $\{b.cantidadUsuarios() = b_0.cantidadUsuarios() + 1 \wedge_L b.saldo(b.cantidadUsuarios()) = 0\}$

2. `proc actualizarSaldo` (in id: $\mathbb{Z}$, in monto: $\mathbb{Z}$, inout b: Berretacoin) :

   `requiere` $\{b = b_0\}$

   `requiere` $\{esUsuarioValido(id)\}$

   `asegura` $\{b.saldo(id) = b_0.saldo(id) + monto\}$

3.2. Justificativo de los procesos:

1. En este proseco, al agregar un usuario se le otorga como identificador un natural que es el siguiente al usuario que se agrego justo antes a él, esto nos permite no tener huecos y futuros problemas a la hora de implementar con las id´s.
2. Este proceso nos interesará más adelante para actualizar el saldo de los compradores y vendedores que participaron en transacciones validadas en un bloque que será agregado a la blockchain.

3.3. Predicados:

1. `pred esUsuarioValido` (id: $\mathbb{Z}$, b = Berretacoin) {

   $id \leq b_0.cantidadUsuarios()$

   }
2. `pred usuariosDistintos` (in $\mathrm{id}_1 : \mathbb{Z}$, in $id_2 : \mathbb{Z}$){

   $id_1 \neq id_2$

}

4. Transacciones

4.1. Funciones auxiliares:

1. `aux Id.transaccion` (transaccion: Transaccion) : $\mathbb{Z} = transaccion[0]$;
2. `aux Comprador` (transaccion: Transaccion) : $\mathbb{Z} = transaccion[1]$;
3. `aux Vendedor` (transaccion: Transaccion) : $\mathbb{Z} = transaccion[2]$;
4. `aux Monto` (transaccion: Transaccion) : $\mathbb{Z} = transaccion[3]$;

4.2. Comentario sobre las funciones auxiliares:

La idea acá es crear funciones auxiliares para el tipo Transacción, y así acceder a los valores dada una entrada de éste tipo y evitar una utilización constante de subíndices que recargue la escritura. Es decir, si quiero acceder al monto de una transacción, en lugar de escribir $b_0.(id.bloque, id.transaccion)[3]$ lo cual es bastante abstracto y difícil de leer, utilizamos estas funciones auxiliares a modo de reemplazos sintácticos para ayudar a que la especificación sea más comprensible.

Otra cosa llamativa de estas funciones auxiliares, es que cada una podría ser un observador, sin embargo evitamos hacerlo así para no tener una sobrecarga de observadores e intentar usar no más de los necesarios.

4.3. Predicados

1. `pred esDeCreacion` (in id.bloque: $\mathbb{Z}$, in id.transaccion: $\mathbb{Z}$, b = Berretacoin) {

   $(id.bloque \leq 3000) \wedge_L Comprador(b_0.transaccion(id.bloque, id.transaccion)) = 0$

   }
2. `pred esTransaccionValida` (in transaccion: Transaccion, in b: Berretacoin) {

   $\text{true} \iff (Monto(transaccion) > 0) \wedge_L (id.comprador, id.vendedor \leq b_0.cantidadUsuarios()) \wedge_L (saldo(id.comprador) \geq Monto(transaccion))$

   }

4.4. Comentario sobre los predicados:

Esta parte nos sirve, al igual que la anterior, para modularizar los predicados y procesos sobre bloques (sección inmediatamente contigua a ésta), y así facilitar la comprensión de la especificación, pues siendo que debe ser correctamente interpretada no consideramos óptimo que se generen dificultades en su lectura.

5. Bloques

5.1. Predicados

1. `pred transaccionesCrecientes` (in id.bloque: $\mathbb{Z}$, b = Berretacoin) {

   $(\forall i : \mathbb{Z})$ (

   $(\forall j : \mathbb{Z})$ (

   $(0 \leq i < j < |b_0.bloque(id.bloque)| \longrightarrow_L$
   $Id.transaccion((b_0.transaccion(id.bloque, i))) < Id.trasaccion(b_0.transaccion(id.bloque, j))$

)

)

   }

2. `pred sonTransaccionesValidas` (in id.bloque: $\mathbb{Z}$, b = Berretacoin) {

   $(\forall i : \mathbb{Z})$ (

   $0 \leq i \leq |b_0.bloque(id.bloque)| - 1 \longrightarrow_L esTransaccionValida(b_0.bloque(id.bloque)[i])$

)

   }

3. `pred esBloqueValido` (in id.bloque: $\mathbb{Z}$, b = Berretacoin) {

   $(|Bloque| < 50) \wedge_L noHayRepetidos(id.bloque) \wedge_L transaccionesCrecientes(id_bloque, b_0) \wedge_L$
   $sonTransaccionesValidas(id.bloque, b_0)$

   }

4. `pred esBloqueCreacion` (in id.bloque: $\mathbb{Z}$, b = Berretacoin) {

   $esBloqueValido(id.bloque, b) \wedge_L (id.bloque < 3000) \longrightarrow_L (esDeCreacion(b_0.bloque(id.bloque)[0]))$

   }

5.2. Comentarios sobre los predicados:

1. Con este predicado, buscamos ver que el orden de los identificadores de las transacciones dentro de un bloque, es estrictamente creciente.

2. Aquí generalizamos para bloques lo hecho en el predicado de la sección anterior para transacciones. La modularización del predicado $esTransaccionValida$ nos permite mayor legibilidad, que es lo que se buscaba.

3. Nuevamente, y en base a lo pedido, especificamos lo que entendemos por bloqueValido.

4. Aquí, por lo puesto en el enunciado, buscamos diferenciar un tipo especial de bloque que es de nuestro interés, por ser el tipo de bloque en el cual se genera una emisión de nuestra criptomoneda.

6. Procesos y predicados pedidos en el enunciado

6.1. agregarBloque

`proc agregarBloque` (inout b: Berretacoin, in bloque: Bloque)
`requiere` $\{b = b_0\}$
`requiere` $\{bloque.id = b_0.cantidadBloques() + 1\}$
`requiere` $\{esBloqueValido(bloque, b) \vee esBloqueCreacion(bloque, b)\}$
`asegura` $\{b = b_0 + bloque\}$
`asegura` $\{(\forall i : \mathbb{Z})\ (0 \leq i < |bloque| \longrightarrow_L actualizarSaldo(Vendedor(bloque[i]), Monto(bloque[i], b))\}$
`asegura` $\{esBloqueCreacion(bloque) \longrightarrow_L$
$(\forall i : \mathbb{Z})\ (1 \leq i < |bloque| \longrightarrow_L actualizarSaldo(Comprador(bloque[i]), -Monto(bloque[i], b)))\}$
`asegura` $\{\neg esBloqueCreacion(bloque) \longrightarrow_L$
$(\forall i : \mathbb{Z})\ (0 \leq i < |bloque| \longrightarrow_L actualizarSaldo(Comprador(bloque[i]), -Monto(bloque[i], b)))\}$

6.2. Sobre agregarBloque:

Lo pensado aquí, es que cada bloque[i] es una transacción. Usamos entonces los métodos que se definieron en 4 y actualizamos los montos una vez el bloque es validado por la blockchain como correcto.

Para el vendedor no es de interés si la transacción es de creación, pues en una transacción de creación (donde el vendedor "minaüna unidad de Berretacoin), se genera un aumento del monto de éste último.

Si tenemos un bloque de creación, por como construimos $esBloqueValido$, podemos omitir el primer subíndice (el correspondiente al valor 0) y actualizar los montos de los compradores asegurandonos que no actualizaremos el del usuario 0, es decir el emisor de moneda.

Por último, unificamos las condiciones de valides del bloque y que sea de creación, pues por como fue definido el segundo, se entiende que es un sub-tipo de bloque válido.

6.3. maximosTenedores

`proc maximosTenedores` (in b: Berretacoin) : seq$\langle\mathbb{Z}\rangle$
`requiere` $\{b = b_0\}$
`asegura` $\{(cantidadUsuarios() = 0 \wedge_L res = []) \vee_L$
$(\exists i : \mathbb{N})\ ((esUsuarioValido(i) \wedge_L i \in res) \longrightarrow_L (\forall j : \mathbb{N})\ (esUsuarioValido(j) \longrightarrow_L b_0.saldo(j) \leq b_0.saldo(i)))$

6.4. montoMedio

`proc montoMedio` (in cadena: Berretacoin) : $\mathbb{Z}$
`requiere` $\{True\}$
`asegura` $\{res = \dfrac{totalTransacciones(cadena)}{cantidadTransacciones(cadena)}\}$

auxiliares

`aux totalTransacciones` (cadena: Berretacoin) : $\mathbb{Z}$ $= \sum_{i=0}^{|cadena.blockchain|-1} \sum_{j=0}^{|cadena.blockchain[i].transacciones|-1} (getMonto(cadena.blockchain[i].transacciones[j]))$;
`aux cantidadTransacciones` (cadena: Berretacoin) : $\mathbb{Z}$ $= \sum_{i=0}^{|cadena.blockchain|-1} \sum_{j=0}^{|cadena.blockchain[i].transacciones|-1} (1)$;

6.5. cotizacionAPesos

`proc cotizacionAPesos` (in cotizaciones: seq$\langle\mathbb{Z}\rangle$, in cadena: Berretacoin) : seq$\langle\mathbb{Z}\rangle$
`requiere` $\{True\}$
`asegura` $\{(\forall i : \mathbb{Z})\ ((0 \leq i < |cotizaciones|) \longrightarrow_L res[i] = cotizaciones[i] * totalTransaccion(cadena.blockchain[i]))\}$

auxiliares

`aux totalTransaccion` (bloq: Bloque) : $\mathbb{Z}$ $= \sum_{i=0}^{|bloq.transacciones|-1} (getMonto(bloq.transacciones[i]))$;

7. Predicados Generales

`pred noHayRepetidos` (in s: $seq\langle T\rangle$) {
$(\forall i : \mathbb{Z})\ ((\forall j : \mathbb{Z})\ (i \neq j \longrightarrow_L s[i] \neq s[j]))$
}

8. Otras ideas

Hacer todo con 4 observadores:
blockchain = secuendia de Bloques

bloque = secuencia de transacciones. donde cada transaccion es un struc $\text{id}_t ransaccion, id_c omprador, id_v endedor, monto$

monedasEmitidas = me da un Z con la cantidad emitida

obs usuarios = secuencia de usuario donde usuario es un struc¡id, saldo¿
Acá debería redefinir las funciones pero es bastante similar. Preguntar si se puede usar esto, ya que haria todo más sencillo.

9. Preguntas:

- ¿Debo agregar un proceso que sea nuevaTransacción?
- ¿Debo especificar de que tipo son los usuarios? ¿Crear, por ejemplo un diccionario, una lista o un struct al cual preguntarle el saldo? ¿No es sobre-especificar, siendo que puedo armar el observador saldo sin necesitar todo eso?
- ¿El bloque es dado de alguna forma? ¿Viene con id?
- ¿puedo usar lo que está en otras ideas?

}

Parte III
TAD completa: