ENTREGABLE 7

7. ESPECIFICACIÓN DE UN RECURSO COMPARTIDO

El ejercicio consiste en **elaborar la especificación formal del recurso compartido ControlAccesoPuente**. El recurso compartido forma parte de un programa concurrente que gestiona los accesos y salida de coches de un puente de un solo carril. El puente tiene dos entradas y dos salidas. Los coches que entran por la entrada sur salen por la norte y viceversa. En cada entrada existe un detector y una barrera. En cada salida existe un detector. El sistema de detección y barreras tiene el interfaz de control `Puente.java`:

```
/**
 * Interfaz de control de acceso al punete de un solo sentido.
 *
 * Hay dos accesos de entrada al puente, uno al norte y otro al sur.
 *
 * Hay dos salidas del puente, una al norte y otra al sur.
 *
 * En las entradas y en las salidas hay detectores. Para detectar que
 * un vehiculo ha llegado a un detector se usan los metodos detectar.
 *
 * Las entradas estan controladas con barreras. Para abrir una barrera
 * se utiliza el metodo abrir.
 */

public class Puente {
   // TODO: introducir atributos para la simulacion (generadores
   // aleatorios, etc.)

   /**
    * Enumerado con los identificadores de las entradas
    */
   public static enum Entrada { N, S }

   /**
    * Enumerado con los identificadores de las salidas
    */
   public static enum Salida { N, S }

   /**
    * El thread que invoque la operacion detectar queda bloqueado
    * hasta que un coche llegue el detector indicado, en ese momento
    * el thread termina de ejecutar el metodo
    */
   static public void detectar(Entrada e) {
         // TODO: elaborar el codigo de simulacion
   }
```

```
   /**
    * El thread que invoque la operacion detectar queda bloqueado
    * hasta que un coche llegue el detector indicado, en ese momento
    * el thread termina de ejecutar el metodo.
    */
   static public void detectar(Salida s) {
         // TODO: elaborar el codigo de simulacion
   }

   /**
    * Al invocar la operacion se abrira la barrera indicada, se
    * permite la entrada de un coche al puente y se cierra la barrera
    * indicada. El tiempo que tarda en ejecutarse el metodo abrir
    * coincide con el tiempo que tarda en realizarse toda la actividad
    * (abrir-pasar-cerrar).
    */
   static public void abrir(Entrada e) {
         // TODO: elaborar el codigo de simulacion
   }
}
```

El objetivo del programa concurrente es controlar la entrada y salida de vehículos de tal forma que jamás haya en el puente dos coches que pretendan dirigirse en sentido contrario. Se ha decidido un diseño en el que existe un proceso por cada entrada y un proceso por cada salida. Los procesos comparten un recurso compartido que les sirve para comunicarse. A continuación se muestra el interfaz del recurso compartido que hay que especificar (`ControlAccesoPuente.java`):

```
public class ControlAccesoPuente {

   public ControlAccesoPuente() {
   }

   /**
    * Incrementa el numero de coches en el puente que han entrado por
    * la entrada e. Si los coches en el puente van en el sentido
    * opuesto entonces el proceso que lo invoque debe bloquear.
    */
   public void solicitarEntrada(Puente.Entrada e) {
   }

   /**
    * Decrementa el numero de coches en el puente.
    */
   public void avisarSalida(Puente.Salida s) {
   }
}
```

y el programa concurrente `CC_07_Puente.java`:

```
/**
 * Programa concurrente para el control del acceso al puente de un
 * solo sentido.
 */
public class CC_07_Puente {

   static private class ControlEntrada extends Thread {
         private Puente.Entrada e;
         private ControlAccesoPuente cap;

         public ControlEntrada(Puente.Entrada e,
                                     ControlAccesoPuente cap) {
               this.e = e;
               this.cap = cap;
         }

         public void run() {
               while(true) {
                     Puente.detectar(this.e);
                     cap.solicitarEntrada(this.e);
                     Puente.abrir(this.e);
               }
         }
   }

   static private class AvisoSalida extends Thread {
         private Puente.Salida s;
         private ControlAccesoPuente cap;

         public AvisoSalida(Puente.Salida s,
                                     ControlAccesoPuente cap) {
               this.s = s;
               this.cap = cap;
         }

         public void run() {
               while(true) {
                     Puente.detectar(this.s);
                     cap.avisarSalida(this.s);
               }
         }
   }
```

```
  public static final void main(final String[] args)
              throws InterruptedException
  {
        ControlAccesoPuente cap;
        ControlEntrada ceN, ceS;
        AvisoSalida asN, asS;

        cap = new ControlAccesoPuente();
        ceN = new ControlEntrada(Puente.Entrada.N, cap);
        ceS = new ControlEntrada(Puente.Entrada.S, cap);
        asN = new AvisoSalida(Puente.Salida.N, cap);
        asS = new AvisoSalida(Puente.Salida.S, cap);

        ceN.start();
        ceS.start();
        asN.start();
        asS.start();
  }
}
```

EJERCICIOS DE ESPECIFICACIÓN

Dado el siguiente **CTAD**

C-TAD MiCTAD
TIPO: MiCTAD $= \mathit{Indice} \to \mathbb{B}$
TIPO: $\mathit{Indice} = \{0, 1\}$
INICIAL: $\forall\, i \in \mathit{Indice} \bullet \neg\text{self}(i)$
INVARIANTE: $\neg\text{self}(0) \vee \text{self}(1)$

CPRE: $\neg\text{self}(0) \wedge \neg\text{self}(1)$
uno()
POST: $\mathit{self} = \text{self}^{pre} \oplus \{1 \mapsto \mathit{Cierto}\}$

CPRE: $\neg\text{self}(0) \wedge \text{self}(1)$
dos()
POST: $\mathit{self} = \text{self}^{pre} \oplus \{0 \mapsto \mathit{Cierto}\}$

CPRE: $\text{self}(1)$
tres()
POST: $\mathit{self} = \text{self}^{pre} \oplus \{0 \mapsto \mathit{Falso}\} \oplus \{1 \mapsto \mathit{Falso}\}$

Asumiendo que tenemos tres procesos que invocan repetidamente las operaciones *uno*(), *dos*() y *tres*() del recurso compartido. Se pide marcar la afirmación correcta:

(a) Podría llegar a violarse la invariante.

(b) El sistema podría quedarse bloqueado.

(c) El sistema nunca quedará bloqueado y siempre se cumplirá la invariante.

(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

Suponiendo que hay tres procesos *t1*, *t2* y *t3*, que cada uno ejecuta repetidamente las operaciones *uno*, *dos*, *tres* del CTAD que se especifica a continuación:

TIPO: MiCTAD $= (a : \mathbb{B} \times b : \mathbb{B})$
INICIAL: $\mathit{self} = (\text{Falso}, \text{Falso})$
INVARIANTE: $\neg\text{self}.a \vee \neg\text{self}.b$

CPRE: $\neg \mathit{self}.a$
uno()
POST: $\mathit{self}^{pre} = (c, d) \wedge \mathit{self} = (c, \neg d)$
CPRE: $\mathit{self}.a \wedge \neg \mathit{self}.b$
dos()
POST: $\mathit{self}^{pre} = (c, d) \wedge \mathit{self} = (c, \neg d)$
CPRE: $\neg\text{self}.b$
tres()
POST: $\mathit{self}^{pre} = (c, d) \wedge \mathit{self} = (\neg c, d)$

Se pide marcar la afirmación correcta:
(a) El programa podría acabar en interbloqueo pero nunca se violaría la invariante
(b) El programa podría violar la invariante pero nunca acabaría en interbloqueo
(a) El programa podría violar la invariante y además acabar en interbloqueo
(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

Dado el siguiente **CTAD**[1]

TIPO: Mi-CTAD $= (a : \mathbb{N} \times b : \mathbb{B})$
INICIAL: $self = (0, \text{Cierto})$
INVARIANTE: $0 \leq \text{self}.a \leq 1 \wedge (\text{self} = 1 \vee self.b)$

CPRE: Cierto
uno()
POST: $self^{pre} = (c, d) \wedge self = (c, \text{Cierto})$

CPRE: $self.a = 0 \wedge self.b$
dos()
POST: $self^{pre} = (c, d) \wedge self = (c + 1, \neg d)$

CPRE: self.b
tres()
POST: $self^{pre} = (c, d) \wedge self = (0, d)$

Asumiendo que tenemos tres procesos que invocan repetidamente las operaciones *uno*(), *dos*() y *tres*() del recurso compartido. Se pide marcar la afirmación correcta:

(a) Podría llegar a violarse la invariante.

(b) El sistema podría quedarse bloqueado.

(c) El sistema nunca quedará bloqueado y siempre se cumplirá la invariante.

(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

Dado el siguiente **CTAD** (sólo se muestran las partes necesarias):

TIPO: T4 $= (a : \mathbb{B} \times b : \mathbb{B})$
INICIAL: $self = (falso, falso)$
CPRE: $\neg(self.a)$
x()
POST: $self^{pre} = (ae, be) \wedge self = (\neg ae, be)$
CPRE: $self.a$
y()
POST: $self^{pre} = (ae, be) \wedge self = (\neg ae, \neg be)$

Se pide marcar la afirmación correcta:

(a) El recurso puede pasar, a lo sumo, por 3 estados diferentes.

(b) Si el sistema solo contiene (además de un recurso de este tipo) un proceso que intenta ejecutar $x()$ repetidamente y otro que intenta ejecutar $y()$ repetidamente, ambos ejecutarán dichas acciones en alternancia estricta.

(c) El recurso podría estar en un estado dado y, tras ejecutarse una sola de sus acciones, continuar en el mismo estado.

(d) Si el sistema solo contiene (además de un recurso de este tipo) un proceso que intenta ejecutar $x()$ repetidamente y otro que intenta ejecutar $y()$ repetidamente, el sistema podría acabar en interbloqueo.

(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

Dado siguiente **CTAD** (el parámetro de la operación *peligro* es un dato de entrada de tipo $\mathbb{B}$).

C-TAD *Peligro*
TIPO: $Peligro = p : \mathbb{B} \times o : \mathbb{N}$
INICIAL: self $= (false, 0)$

INVARIANTE: self.$p \vee$ self.$o \leq 5$

CPRE: Cierto
peligro(x)
POST: self.$p = x \wedge$ self.$o =$ self.o

CPRE: $\neg$self.$p \wedge$ self.$o < 5$
entrar()
POST: $\neg$self.$p \wedge$ self.$o =$ self.$o + 1$

CPRE: self.$o > 0$
salir()
POST: self.$p =$ self.$p \wedge$ self.$o =$ self.$o - 1$

Se pide marcar la afirmación correcta.

(a) Siempre se cumplirá la invariante.

(b) Podría llegar a violarse la invariante.

Dada la siguiente implementación del **CTAD** de la pregunta anterior:

```
class Peligro {
  private Semaphore p =
    new Semaphore(1);
  private Semaphore o =
    new Semaphore(5);

  public void peligro(boolean x) {
    if (x)
      o.signal();
    else
      o.await();
  }

  public void entrar() {
    o.await();
    p.await();
    p.signal();
  }

  public void salir() {
    o.signal();
  }
}
```

Supóngase se realizan simultáneamente las siguientes siete llamadas por sendos procesos: `peligro(true)`, `entrar()`, `entrar()`, `entrar()`, `entrar()`, `entrar()` y `entrar()` y supóngase que no se realiza ninguna otra llamada al recurso compartido.

Se pide marcar la afirmación correcta.

(a) Ningún proceso estará bloqueado.

(b) Un proceso estará bloqueado.

(c) Más de un proceso estará bloqueado.

Suponiendo que hay tres procesos que ejecutan repetidamente las operaciones *uno*, *dos*, *tres* del CTAD que se especifica a continuación:

TIPO: MiCTAD $= (a : \mathbb{B} \times b : \mathbb{B})$
INICIAL: *self* $=$ (Falso, Falso)
INVARIANTE: $\neg self.a \vee \neg self.b$

CPRE: $\neg self.a$
uno()

POST: $self^{pre} = (a, b) \wedge self = (a, \neg b)$
CPRE: $self.a \vee self.b$
dos()
POST: $self^{pre} = (a, b) \wedge self = (\neg a, \neg b)$
CPRE: Cierto
tres()
POST: $self^{pre} = (a, b) \wedge self = (\text{Cierto}, b)$

Se pide marcar la afirmación correcta:
(a) El programa no podría llegar a violar el invariante
(b) El programa podría llegar a violar el invariante
(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

La compañía espacial *SpaceÑ* está desarrollando un cohete capaz de regresar al punto de partida y aterrizar de pie. El ajuste fino de la trayectoria se realiza mediante unos alerones, pero para desviaciones más grandes puede recurrir a cuatro pequeños propulsores laterales, que denominaremos por los puntos cardinales N, O, S y E.

El software del cohete dispone de un proceso que mide periódicamente la desviación y la comunica al sistema. Por otra parte, hay cuatro procesos controlando los propulsores laterales. En el momento en el que la desviación en una de las cuatro direcciones supera un cierto umbral dicho proceso da la orden de dar un impulso puntual con el propulsor, con una intensidad proporcional a dicha desviación.

Por ejemplo, el proceso que controla la desviación ejecutaría en bucle la secuencia:

medirdesviacion(dx,dy); *notificar(dx,dy)*;

El proceso encargado de activar el propulsor "norte", ejecutaría en bucle:

detectarN(dy); *activarpropulsorN(dy)*;

Las operaciones *medirdesviacion* y *activarpropulsorN* son métodos de la librería que interactúa con el hardware del cohete (dadas). Las operaciones que formarían parte del recurso mediante el cual interactúan los threads serían *notificar* y *detectarN*.

El recurso *Cohete* almacena las dos componentes del vector de desviación — inicialmente $(0, 0)$. La operación *notificar* no es bloqueante y actualiza dicho vector. La operación *detectarN* bloquea hasta que la segunda componente del vector de desviación supera la constante umbral *DM*, y devuelve el valor absoluto de dicha componente. Las otras tres operaciones de detección son similares, bajo rotación y simetría.

Se pide: completar la especificación del recurso compartido.

C-TAD Cohete
OPERACIONES
ACCIÓN notificar: *TDesviacion[e]* × *TDesviacion[e]*
ACCIÓN detectarN: *TDesviacion[s]*
ACCIÓN detectarO: *TDesviacion[s]*
ACCIÓN detectarS: *TDesviacion[s]*
ACCIÓN detectarE: *TDesviacion[s]*
SEMÁNTICA
DOMINIO:
TIPO: *TDesviacion* = {*0 .. 100*}
TIPO: *Cohete* =
INICIAL:
CPRE:
notificar(dx,dy)
POST:
CPRE:
detectarN(dy)
POST:

Se pide completar la siguiente especificación de un recurso compartido que implementa la adquisición simultánea de pares de tokens adyacentes escogidos de un vector circular de tokens. Las operaciones *adquirir* y *soltar* reciben un índice válido del vector de tokens. La operación *adquirir(i)* corresponde a la adquisición de los tokens en las posiciones i e $i+1$ (o 0 si $i+1$ es igual al número total de tokens). La operación *soltar(i)* devuelve los tokens de las posiciones i e $i+1$.

Un proceso puede invocar a la adquisición de dos tokens adyacentes cuando éstos no estén disponibles, en cuyo caso tendrá que bloquearse hasta que lo estén. Por el contrario, no se permite que un proceso invoque *soltar* sobre tokens que no estén asignados (podéis usar la clásula PRE para expresar esta prohibición).

Se pide especificar un recurso compartido que gestiona un sistema de backups que dispone de varios discos duros para realizar las copias de seguridad. En este recurso interactúan dos tipos de procesos: unos que se encargan de *hacer backups* y otros que se encargan de ir *reponiendo los discos* a medida que éstos se van llenando.

El sistema de backups dispone de *NUM_DISCOS* discos duros con capacidad *MAX_BYTES*. La operación *hacerBackup*(*size*) solicita la realización de un backup de tamaño *size* bytes. El valor del parámetro *size* siempre debe ser menor que *MAX_BYTES*. El sistema calcula en qué disco hacer el backup mediante la operación *size* %*NUM_DISCOS*. Por ejemplo, si $size = 14$ y $NUM\_DISCOS = 5$ el sistema hará el backup en el disco número 4. En caso de se intente realizar un backup y el disco asignado ya haya superado su capacidad, el proceso deberá bloquearse hasta que vuelva a haber espacio en el disco correspondiente. La operación *reponerDisco*(i) deberá bloquearse mientras el disco i no esté lleno. Cuando el disco se haya llenado, es decir, haya superado el tamaño *MAX_BYTES*, *reponerDisco*(i) deberá reponer el disco i indicando que el espacio ocupado del disco i es de 0 bytes. Se ha definido el tipo *PosDisco* para indicar que i siempre estará entre los valores válidos, es decir, $[0..NUM\_DISCOS - 1]$. Se asume, por simplificar, que se puede superar el tamaño máximo del disco en la última operación, es decir, si un disco de tamaño 100, tiene ocupados 80 y le llega un backup de 30, este podrá hacerse aunque se supere la capacidad del disco.

C-TAD SistemaBackup

OPERACIONES

ACCIÓN hacerBackup: $\mathbb{Z}$*[e]*
ACCIÓN reponerDisco: *PosDisco[e]*

SEMÁNTICA
DOMINIO:

TIPO: *SistemaBackup* =

TIPO: *PosDisco* $= 0..NUM\_DISCOS - 1$

INICIAL:

PRE:

CPRE:

hacerBackup(size)

POST:

CPRE:

reponerDisco(i)

POST:

Se pide especificar un recurso compartido que mezcla dos secuencias ordenadas de números enteros para formar una única secuencia ordenada. En este recurso interactúan tres procesos: dos productores que van pasando números de sus secuencias de uno en uno y un consumidor que va extrayendo los números en orden.

Cada recurso será capaz de almacenar, como mucho, un dato de la secuencia que llamaremos "izquierda" y un dato de la secuencia "derecha". Cuando hay datos de ambas secuencias la operación *extraerMenor* tomará el menor de ambos y permitirá que se añada un nuevo dato de la secuencia correspondiente. La operación *insertarIzda(d)* inserta el dato *d* como parte de la secuencia izquierda. Bloquea hasta que el hueco para el dato izquierdo está disponible. La operación *insertarDcha* es análoga.

Por simplificar no hemos tenido en cuenta aquí la terminación de las secuencias, que requiere de operaciones y estado adicionales.

C-TAD OrdMezcla

OPERACIONES

ACCIÓN insertarIzda: $\mathbb{Z}[e]$
ACCIÓN insertarDcha: $\mathbb{Z}[e]$
ACCIÓN extraerMenor: $\mathbb{Z}[s]$

SEMÁNTICA
DOMINIO:

TIPO: $OrdMezcla =$

TIPO: $Lado = Izda \mid Dcha$

INICIAL:

CPRE:

insertarIzda(d)

POST:

CPRE:

extraerMenor(min)

POST:

Se está desarrollando un sistema concurrente de ordenación por burbuja (*bubblesort*). En este, $N-1$ procesos permutadores se encargan de comparar y, si es necesario, permutar pares de elementos adyacentes de un vector de tamaño N. Para evitar situaciones de carrera en el acceso al vector se va a disponer de un recurso con operaciones para reservar o liberar el acceso a posiciones consecutivas en el vector.

Así, el proceso *permutador*(i) ejecutaría, en bucle, la siguiente secuencia:

$$\textit{reservar}(i);\ \langle\ \textbf{si}\ v[i] > v[i+1]\ \textbf{entonces}\ \textit{permutar}(v[i], v[i+1])\ \rangle;\ \textit{liberar}(i);$$

La operación *reservar*(i) impide que otro proceso pueda acceder a las posiciones i e $i+1$. Para ejecutarla, es necesario que ambas posiciones no estén siendo usadas por otro proceso permutador (la llamada bloquearía hasta que esa condición se dé). La operación *liberar*(i) vuelve a dejar disponibles las posiciones i e $i+1$. No se permite que un proceso invoque *liberar* sobre posiciones que no estén reservadas (podéis usar la clásula PRE para expresar esta prohibición).

Se pide: completar el siguiente CTAD, de acuerdo con el comportamiento arriba explicado.[1]

En la especificación formal de un recurso gestor de *lectores/escritores* hay riesgo de inanición de los procesos escritores ya que los lectores podrían monopolizar el acceso al documento compartido adquiriendo y liberando permisos de lectura indefinidamente, sin permitir acceder a ningún escritor.

Se pide: modificar la especificación formal del gestor de lectores/escritores para forzar a que el número de lectores no crezca cuando hay un escritor en espera. Más concretamente, se ha dividido la operación *inicioEscribir* en dos: una primera que declara la intención de escribir por parte de un proceso escritor (*intencionEscribir*) y una segunda que realmente solicita el acceso (*permisoEscribir*). Tendréis que extender el estado del recurso para llevar cuenta de los escritores en espera y modificar el resto de la especificación de acuerdo con estos cambios.

C-TAD GestorLE_2
OPERACIONES
ACCIÓN intencionEscribir:
ACCIÓN permisoEscribir:
ACCIÓN finEscribir:
ACCIÓN inicioLeer:
ACCIÓN finLeer:

SEMÁNTICA
DOMINIO:
TIPO: *GestorLE_2* =

INICIAL:

INVARIANTE:

CPRE:
intencionEscribir()
POST:

CPRE:
permisoEscribir()
POST:

CPRE:
finEscribir()
POST:

CPRE:

inicioLeer()

POST:

CPRE:

finLeer()

POST:

Se está desarrollando una IA para competir en el campeonato del mundo del popular juego de mesa *Los nómadas que cantan*. Los desarrolladores han ideado un recurso compartido para representar las materias primas que tiene el jugador, siendo estas materias primas cereal, agua y madera.

Hay una operación no bloqueante por cada materia prima para que el jugador coja esa materia prima: **cargarCereal**, **cargarAgua** y **cargarMadera**. El jugador puede llevar a lo sumo una unidad de cada materia prima pero no puede llevar cereal y madera a la vez, si carga una pierde la otra.

Por otro lado, hay dos operaciones que consumen materias primas: **avanzar** (consume cereal y agua) y **reparar** (consume agua y madera). Estas operaciones son bloqueantes hasta que se disponga de las materias primas necesarias.

Se pide: especificar el recurso compartido teniendo en cuenta que bastaría un booleano por cada materia prima para indicar que el jugador tiene dicha materia (porque la ha cargado) o no tiene dicha materia (porque acaba de empezar el juego o la ha consumido avanzando o reparando).

C-TAD MateriasPrimas
OPERACIONES
ACCIÓN cargarCereal:
ACCIÓN cargarAgua:
ACCIÓN cargarMadera:
ACCIÓN avanzar:
ACCIÓN reparar:
SEMÁNTICA
DOMINIO:
TIPO: *MateriasPrimas* =
INICIAL:
INVARIANTE:

CPRE:
cargarCereal
POST:

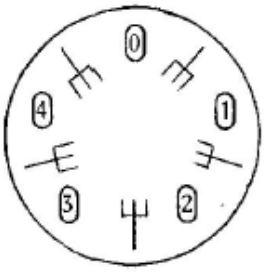
CPRE:
cargarAgua
POST:

CPRE:
cargarMadera
POST:

CPRE:	
avanzar	
POST:	
CPRE:	
reparar	
POST:	

Lo que veis a continuación es la descripción original de Edsger W. Dijkstra del problema de los *Dining Philosophers*:

> *La vida de un filósofo consiste en alternar pensar y comer en una especie de bucle infinito. Cinco filósofos, numerados del 0 al 4, viven en una casa con una mesa en la que cada filósofo tiene su sitio asignado:*



> *Su único problema, al margen de los filosóficos, es que su menú consiste en un tipo complicado de espagueti que es necesario comer con dos tenedores. Hay un único tenedor a cada lado de cada plato. Eso no es un problema. Sin embargo, como consecuencia, no puede haber dos filósofos comiendo a la vez uno al lado del otro.*

Nosotros hemos decidido representar el problema en forma de un recurso compartido m (de mesa) con dos operaciones: $m.cogerTenedores(i)$ y $m.soltarTenedores(i)$. Esas serán las operaciones que el filósofo i ejecutará antes y después de comer, de forma que tendrá que bloquear en $m.cogerTenedores(i)$ cuando alguno de los tenedores a los lados de su plato estén siendo usados (por el filósofo $(i+1) \bmod 5$ o por el filósofo $(i-1) \mod 5$).

Se pide completar el recurso compartido sobre el esqueleto en la siguiente página.

Nota: se sugiere que el dominio del recurso sea una función parcial que hable de los tenedores libres. Las funciones parciales formalizan las tablas. La declaración del tipo de una función parcial de A en B es $A \nrightarrow B$. Si self es una función parcial del tipo $A \nrightarrow B$, la expresión self(a) representa el valor de para la *clave* a, la expresión self $\oplus \{a \mapsto b\}$ representa la *tabla* self cambiando la entrada de la *clave* a por el valor b y la expresión dom self representa el conjunto de claves de self.

C-TAD Mesa

OPERACIONES

ACCIÓN cogerTenedores: $\mathbb{N}[e]$
ACCIÓN soltarTenedores: $\mathbb{N}[e]$

SEMÁNTICA
DOMINIO:

TIPO: *Mesa* =

INVARIANTE:

INICIAL:

PRE:

CPRE:

cogerTenedores(i)

POST:

PRE:

CPRE:

soltarTenedores(i)

POST: