REPASO PARCIAL 2

1. Dada la siguiente especificación formal de un recurso compartido *Peligro*. Se pide: Completar la implementación de este recurso mediante paso de mensajes:

C-TAD *Peligro*
OPERACIONES
ACCIÓN *avisarPeligro*: $\mathbb{B}[e]$
ACCIÓN *entrar*:
ACCIÓN *salir*:

SEMÁNTICA
DOMINIO:
TIPO: $Peligro = (p : \mathbb{B} \times o : \mathbb{N})$
INICIAL: $\text{self} = (false, 0)$

INVARIANTE: $\text{self}.o \leq 5$

CPRE: Cierto
***avisarPeligro*(x)**
POST: $\text{self}.p = x \wedge \text{self}.o = \text{self}^{pre}.o$

CPRE: $\neg\text{self}.p \wedge \text{self}.o < 5$
***entrar*()**
POST: $\neg\text{self}.p \wedge \text{self}.o = \text{self}^{pre}.o + 1$

CPRE: $\text{self}.o > 0$
***salir*()**
POST: $\text{self}.p = \text{self}^{pre}.p \wedge \text{self}.o = \text{self}^{pre}.o - 1$

2. A continuación mostramos la especificación formal de un recurso gestor *Misil*. Se pide: Completar la implementación de este recurso mediante paso de mensajes. NOTA: Podéis usar el método Math.abs(x) para calcular el valor absoluto de un número, |x|:

C-TAD Misil
OPERACIONES
ACCIÓN notificar: $\mathbb{Z}[e]$
ACCIÓN detectarDesviacion: $TUmbral[e] \times \mathbb{Z}[s]$

SEMÁNTICA
DOMINIO:
TIPO: $TUmbral = [0..100]$
TIPO: $Misil = \mathbb{Z}$

INICIAL: $\text{self} = 0$

CPRE: *Cierto*
notificar(desv)
POST: $\text{self} = desv$

CPRE: $|\text{self}| > umbral$
detectarDesviacion(umbral,d)
POST: $self = \text{self}^{pre} \wedge d = \text{self}^{pre}$

3. El siguiente recurso compartido forma parte de un algoritmo paralelo de ordenación por mezcla. Permite mezclar dos secuencias ordenadas de números enteros para formar una única secuencia ordenada. En este recurso interactúan solo tres procesos: dos productores (izquierdo y derecho) que van pasando números de sus secuencias de uno en uno y un consumidor que va extrayendo los números en orden.

C-TAD: OrdMezcla

OPERACIONES:
ACCIÓN: insertar: Lado[e] x Z [e]
ACCIÓN: extraerMenor: Z[s]

SEMÁNTICA:
DOMINIO:
TIPO: OrdMezcla = { haydato: Lado → B x dato: Lado → Z }
TIPO: Lado = Izda | Dcha

INICIAL: $\forall i \in Lado \cdot \neg$self.hayDato(i)

CPRE: ¬self.hayDato(l)
insertar(l, d)
POST: $self^{PRE}$ = (hay, dat) ∧ self = <hay ⊕ {l → Cierto} ∧ dat ⊕ {l → d}>

CPRE: self.hayDato(Izda) ∧ self.hayDato(Dcha)
extraerMenor(min)
POST: $self^{PRE}$ = (hay, dat) ∧
(dat(Izda) ≤ dat(Dcha) ∧ min ⇒ dat(Izda) ∧ self = < hay ⊕ {Izda → Falso}, dat>) ∧
(dat(Dcha) ≤ dat(Izda) ∧ min ⇒ dat(Dcha) ∧ self = < hay ⊕ {Dcha → Falso}, dat>)

La operación insertar(lado, dato) inserta dato en el lado correspondiente, bloqueando si ese hueco no está disponible. 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. Por concisión, no hemos considerado el problema de la terminación de las secuencias.

Se pide: Completar la implementación de este recurso compartido mediante paso de mensajes.

4. Una pequeña variación del típico problema del productor-buffer-consumidos es el llamado “buffer de pares e impares”. La idea es que en la operación de extracción se permite decir si queremos retirar un número par o un número impar. Se pide implementar el recurso compartido usando paso de mensajes síncrono, mediante la librería JCSP, a partir de la siguiente especificación:

C-TAD BufferPI

OPERACIONES
ACCIÓN Poner: *Tipo_Dato[e]*
ACCIÓN Tomar: *Tipo_Paridad[e]* × *Tipo_Dato[s]*

SEMÁNTICA
DOMINIO:
TIPO: $Tipo\_Buffer\_PI = \text{Secuencia}(Tipo\_Dato)$
$Tipo\_Paridad = par|impar$
$Tipo\_Dato = \mathbb{N}$
INVARIANTE: $\forall b \in Tipo\_Buffer\_PI \bullet \text{Longitud}(b) \leq MAX$

DONDE: $MAX = ...$
INICIAL: $\text{Longitud}(self) = 0$

CPRE: *El buffer no está lleno*
CPRE: $\text{Longitud}(self) < MAX$
Poner(d)
POST: *Añadimos un elemento al buffer*
POST: $l = \text{Longitud}(self^{pre}) \wedge \text{Longitud}(self) = l+1 \wedge self(l+1) = d^{pre} \wedge self(1..l) = self^{pre}$

CPRE: *El buffer no está vacío y el primer dato preparado para salir es del tipo que requerimos*
CPRE: $\text{Longitud}(self) > 0 \wedge Concuerda(self(1), t)$
DONDE: $Concuerda(d, t) \equiv (d \bmod 2 = 0 \leftrightarrow t = par)$
Tomar(t, d)
POST: *Retiramos el primer elemento del buffer*
POST: $l = \text{Longitud}(self^{pre}) \wedge self^{pre}(1) = d \wedge self = self^{pre}(2..l)$

La solución consistirá en una clase Gestor_PI con la siguiente estructura:

```
static class Gestor_PI implements CSProcess {
  private static final int PAR = 0;
  private static final int IMPAR = 1;
  // A: declaraciones
  public Gestor_PI() {
    // B: inicializacion
  }
  public void poner(int d) {
    // C: op. poner
  }
  public void tomar(int t) {
    // D: op. tomar
  }
    public void run() {
    // E: cod. servidor
  }
}
```

5.

Supóngase un objeto `c` de una clase `C` en la que se han definido los métodos `o` y `p`, declarados como **`synchronized`**. Supóngase también que un thread `t` entra en un bucle infinito *mientras* ejecuta el código de `c.o()`.

Se pide señalar la respuesta correcta.

(a) Cualquier otro thread distinto de `t` que intente ejecutar `c.p()` quedará bloqueado para asegurar la exclusión mutua.

(b) Cualquier otro thread distinto de `t` que intente ejecutar `c.p()` no quedará bloqueado porque la exclusión mutua afecta sólo a las llamadas a `c.o()`.

(c) Cualquier otro thread distinto de `t` que intente ejecutar `c.p()` no quedará bloqueado porque la exclusión mutua no se tiene en cuenta en el caso de un bucle infinito.

6.

Supóngase que dos threads ejecutan los siguientes códigos:

```
while (true) {
   S11;
   ch1.in.read();
   S12;
   ch2.out.write(null);
   S13;
}
```

```
while (true) {
  ch1.out.write(null);
  ch2.in.read();
  S2;
}
```

Suponemos que no hay otros threads que afecten al problema aparte de los mostrados. **Se pide** señalar la respuesta correcta.

(a) `S11` y `S2` no pueden ejecutar simultáneamente.

(b) `S12` y `S2` no pueden ejecutar simultáneamente.

(c) `S13` y `S2` no pueden ejecutar simultáneamente.

7.

¿Debería permitirse a un *thread* invocar una operación de `await` sobre un objeto de clase `Monitor.Cond` generado a partir de un objeto de la clase `Monitor` sin previamente haber invocado el método `enter`?

(a) Sí.

(b) No.

8. Partimos de la siguiente especificación formal de un recurso compartido:

OPERACIONES

ACCIÓN A:
ACCIÓN B:
ACCIÓN C:

SEMÁNTICA
DOMINIO:
TIPO: $Binario = (p : \mathbb{B} \times q : \mathbb{B})$

INICIAL: $self.p \wedge self.q$

CPRE: Cierto
A()
POST: $self.p \wedge (self.q = self^{pre}.q)$

CPRE: Cierto
B()
POST: $\neg self.p \wedge self.q$

CPRE: $self.p \wedge self.q$
C()
POST: $\neg self.p \wedge \neg self.q$

Alguien ha decidido implementarlo mediante monitores de la siguiente manera:

```
class Binario {
    private boolean p = true;
    private boolean q = true;
    private Monitor mutex = new Monitor();
    private Monitor.Cond cond_p = mutex.newCond();
    private Monitor.Cond cond_q = mutex.newCond();

    void a() {
        mutex.enter();
        p = true;
        cond_p.signal();
        mutex.leave();
    }
    void b() {
        mutex.enter();
        p = false;
        q = true;
        cond_q.signal();
        mutex.leave();
    }
    void c() {
        mutex.enter();
        if (!p) { cond_p.await(); }
        if (!q) { cond_q.await(); }
        p = false;
        q = false;
        mutex.leave();
    }
}
```

Se pide señalar la respuesta correcta:

(a) Es una implementación correcta del recurso compartido.

(b) El código puede provocar que se ejecute `c()` en un estado en el que no se cumple su CPRE.

(c) El código de `c` está incompleto: falta el código de desbloqueos.

9.

¿Debería permitirse a un *thread* invocar una operación de `await` sobre un objeto de clase `Monitor.Cond` generado a partir de un objeto de la clase `Monitor` sin previamente haber invocado el método `enter`?

(a) **Sí.**

(b) **No.**

10.

Se define una clase *servidor* P y dos clases *cliente* P1 y P2 que se comunican a través de los canales de comunicación petA y petB (se muestran las partes relevantes del código para este problema).

```
class P implements CSProcess {
  public void run() {
    // Estado del recurso
    boolean q = true;
    boolean r = true;

    final int A = 0;
    final int B = 1;

    final Guard[] entradas =
      {petA.in(), petB.in()};

    final Alternative servicios =
      new Alternative (entradas);

    final boolean[] sincCond =
      new boolean[2];

    while (true) {
      sincCond[A] = q || r;
      sincCond[B] = q && r;

      switch (servicios.fairSelect(sincCond)) {
        case A:
          petA.in().read();
          q = !q;
          break;
        case B:
          petB.in().read();
          r = !r;
          break;
      }
    }
  }
}
```

```
class P1 implements CSProcess {
  public void run() {
    while (true) {
      petA.out().write(null);
    }
  }
}

class P2 implements CSProcess {
  public void run() {
    while (true) {
      petB.out().write(null);
    }
  }
}
```

Dado un programa concurrente con tres procesos p, p1 y p2 de las clases P, P1 y P2.

Se pide decir cuál de las siguientes afirmaciones es la correcta:

(a) Es seguro que los tres procesos se van a bloquear.

(b) Es seguro que p2 acabará bloqueándose, pero p y p1 podrían seguir ejecutando indefinidamente.

(c) Es posible que p1 se bloquee pero en ese caso p y p2 podrían seguir ejecutando indefinidamente.

(d) Ninguna de las respuestas anteriores.

11.

Está a punto de inaugurarse el primer centro comercial en el que la compra la realizan robots: e-Qea. Los compradores envían sus listas de la compra y los robots inician el recorrido. El centro comercial lo componen N (un número fijo) secciones adyacentes. Los robots inician la compra entrando en la sección 0, avanzan a la sección adyacente (0, 1, ...) cuando han acumulado los productos requeridos de esa sección y terminan saliendo de la sección $N-1$.

El centro e-Qea tiene un problema estructural (literalmente): cada sección soporta un peso máximo (P). Esto significa que cuando un robot con un determinado peso (p) quiere avanzar, debe esperar hasta que el incremento de peso que provoca no ponga en peligro la estructura (si el peso actual de la sección adyacente es *peso* entonces un robot con peso p no puede avanzar si $peso + p > P$).

El siguiente recurso compartido gestiona el movimiento de robots por e-Qea:

C-TAD ControleQea

OPERACIONES

ACCIÓN avanzar: $TipoSección[e] \times \mathbb{N}[e]$

SEMÁNTICA
DOMINIO:
TIPO: $ControleQea = TipoSección \nrightarrow \mathbb{N}$
TIPO: $TipoSección = \{0, 1, \ldots, N\}$
INVARIANTE: $\forall s \in TipoSección \bullet self(i) \leq P$

INICIAL: $\forall s \in TipoSección \bullet self(i) = 0$

PRE: Cierto
CPRE: $s < N \Rightarrow \text{self}(s) + p \leq P$
avanzar(s,p)
POST: $s = 0 \Rightarrow \text{self} = \text{self}^{pre} \oplus \{s \mapsto \text{self}^{pre}(s) + p\}$
$s > 0 \Rightarrow \text{self} = \text{self}^{pre} \oplus \{s-1 \mapsto \text{self}^{pre}(s-1) - p, s \mapsto \text{self}^{pre}(s) + p\}$

Los *threads* que controlan los robots ejecutan la operación *avanzar*(s, p) cuando el robot quiere avanzar de la sección $s-1$ a la sección s portando un peso p (con $s = 0$ se indica la entrada al centro y con $s = N$ la salida del mismo).

Se pide implementar el recurso compartido en Java usando la librería JCSP.

12. Se define una clase servidor P y tres clases cliente P1, P2 y P3 que se comunican a través de los canales de comunicación petA, petB y petC (se muestran las partes relevantes del código para este problema).

```
class P implements CSProcess {
  public void run() {
    boolean q = false;
    boolean r = false;
    boolean s = false;
    final int A = 0;
    final int B = 1;
    final int C = 2;
    final Guard[] entradas =
        {petA.in(), petB.in(),petC.in()};
    final Alternative servicios =
        new Alternative (entradas);
    final boolean[] sincCond =
        new boolean[3];

    while (true) {
      sincCond[A] = !(q || r);
      sincCond[B] = !(q && r);
      sincCond[C] = !s;
      int sel = servicios.fairSelect(sincCond);
      switch (sel) {
      case A:
        petA.in().read();
        q = !q;
        break;
      case B:
        petB.in().read();
        r = !r;
        break;
      case C:
        petC.in().read();
        break;
      }
    }
  }
}
```

```
class P1 implements CSProcess {
  public void run() {
    while (true) {
      petA.out().write(null);
    }
  }
}

class P2 implements CSProcess {
  public void run() {
    while (true) {
      petB.out().write(null);
    }
  }
}

class P3 implements CSProcess {
  public void run() {
    while (true) {
      petC.out().write(null);
    }
  }
}
```

Dado un programa concurrente con tres procesos p, p1, p2 y p3 de las clases P, P1, P2 y P3.
Se pide marcar cuál de las siguientes afirmaciones es la correcta:
(a) Es seguro que los cuatro procesos se van a bloquear.
(b) Es seguro que p1 y p2 acabar´an bloque´andose, pero p y p3 podrían seguir ejecutando indefinidamente.
(c) Es posible que p3 se bloquee pero en ese caso p, p1 y p2 podrían seguir ejecutando indefinidamente.
(d) Ninguna de las otras respuestas.

13. Dado el sistema de la pregunta 5 implementado con JCSP. Se pide decir cuál de las siguientes afirmaciones es la correcta:
(a) El array sincCond contiene la evaluación de las CPREs de las operaciones A, B y C.
(b) Cuando fairSelect devuelve un valor sel se cumple que sincCond[sel] == true.
(c) Ambas son correctas