Metodología para la programación de recursos compartidos en Ada
(PARTE I: OBJETOS PROTEGIDOS)

Unidad de Programación

LSIIS

Rev: 394, 15 de noviembre de 2005

Índice

1. Introducción

Una buena metodología de programación atiende a los objetivos principales de corrección, claridad y eficiencia. La programación estructurada, incluyendo el empleo de tipos abstractos de datos, constituye la metodología fundamental recomendada para el desarrollo de programas secuenciales.

El desarrollo de programas concurrentes exige algunos elementos adicionales con respecto a la programación secuencial. Estos elementos son fundamentalmente nuevos mecanismos abstractos para expresar comunicación y sincronización, y que posean unas buenas cualidades estructurales y de abstracción, de manera que constituyan una ampliación coherente de los mecanismos de programación secuencial, para los que sigan siendo válidas las recomendaciones fundamentales de la programación estructurada, es decir:

- Desarrollo mediante refinamiento progresivo.
- Desarrollo simultáneo de las estructuras de control y de datos.
- Empleo de la idea de abstracción para dividir sistemas complejos.

Una metodología general de desarrollo de programas concurrentes, siguiendo esas recomendaciones, podría seguir los siguientes pasos, que son en el fondo independientes del método final de implementación:

1. Identificar los procesos.
2. Plantear el flujo de ejecución de cada proceso.
3. Identificar la interacción entre procesos.
4. Elegir una forma de gestionar la interacción y plasmarla en una especificación de la misma, centrada en la figura de un recursos compartido.
5. Refinar la gestión de la interacción, usando el mecanismo elegido.
6. Analizar y probar la corrección de la solución obtenida.

Los puntos 5 y 6 han de entenderse como elementos de un ciclo por el que se puede pasar una o más veces. Habitualmente, la aplicación perfecta de los pasos 1 a 5 nos asegurará una solución correcta *sólo en cuanto a las propiedades de seguridad*, pero con posibles problemas de vivacidad. Tras analizar dichos problemas (paso 6) deberemos volver al paso 5 para rehacer la programación de la interacción *sin perder las propiedades de seguridad ya aseguradas*. La figura 1 recoge los pasos anteriores, a otro nivel de granularidad, de modo gráfico.

1.1. Motivación y contenidos

El resto de este documento se dedica a describir el mecanismo de objetos protegidos, y la manera de programarlos por refinamiento progresivo de forma sistemática.

Durante unos cuantos cursos se usó el mecanismo de *monitores* como base de una metodología para la programación de recursos compartidos. Si bien nadie discute la claridad y simplicidad de esta propuesta como ventajas para la enseñanza de la concurrencia, la práctica no ha jugado a su favor, ya que son pocos los lenguajes de programación que han optado por realizarla fielmente.[1].

Al optar por el lenguaje Ada como medio común para la enseñanza de la programación, nos alejamos ligeramente de los mecanismos clásicos – monitores, *CSP* – al adoptar los propuestos en este lenguaje de programación – *objetos protegidos*, *rendez-vous* – pero los conceptos subyacentes siguen siendo muy similares.

[1] En el ámbito del lenguaje Java se usa a menudo el término *monitor*, pero lo que este lenguaje ofrece se puede ver como una realización muy limitada de la propuesta original [Hoa74, Bri75].

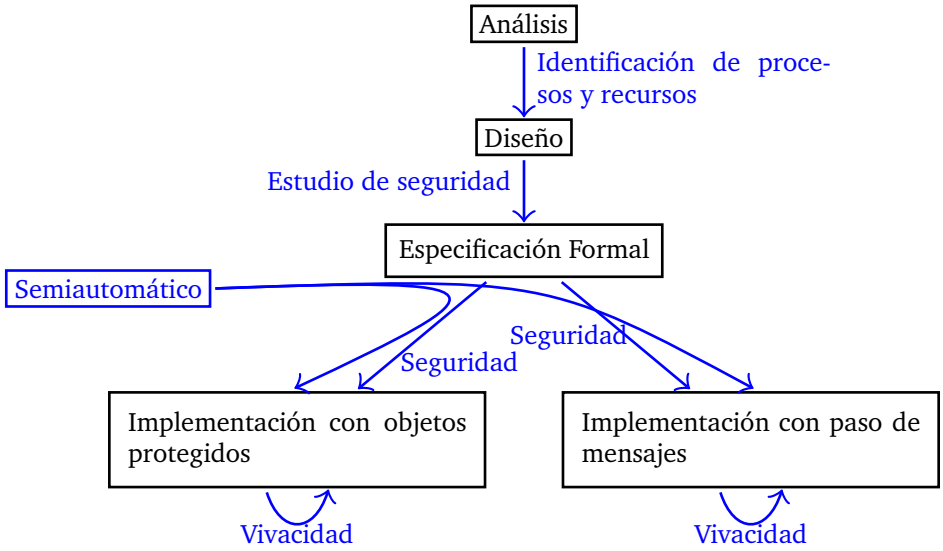

Figura 1: Un esquema de un metodo de desarrollo de programas concurrentes

En estas notas se exponen unas recomendaciones metodológicas para el desarrollo de programas concurrentes empleando el mecanismo de objetos protegidos que Ada proporciona para la comunicación y sincronización de procesos. No obstante, el énfasis se hará en la resolución de problemas de concurrencia de forma correcta, más que en profundizar en aspectos particulares del lenguaje Ada. El objetivo es que todas las ideas expuestas a continuación sean de fácil aplicación al usar otros lenguajes de programación.

2. Concurrencia basada en memoria común

La programación concurrente se ocupa de las descripción de programas en los que varias operaciones o cómputos pueden realizarse simultáneamente. Con el paradigma de programación imperativa, un programa concurrente aparece como un conjunto de *procesos* que se pueden ejecutar simultáneamente, cada uno de los cuales tiene su propia línea de flujo de ejecución. Es decir, cada proceso aparece como un programa secuencial separado.

2.1. Comunicación y sincronización

Los distintos procesos de un programa concurrente han de colaborar en la obtención de los resultados deseados. Para ello necesitan, además de los elementos generales de cualquier programa imperativo, mecanismos adecuados para comunicarse y sincronizarse entre ellos. La comunicación implica que varios procesos pueden utilizar unos mismos datos. La sincronización implica que operaciones de distintos procesos se realizan en un orden adecuado.

Disponiendo de memoria común a varios procesos, la manera más directa de realizar la comunicación es mediante *variables compartidas*, globales, comunes a todos ellos. Para la sincronización se necesitan mecanismos especiales. Si sólo se cuenta con la facilidad de acceso a variables globales, la sincronización habría de realizarse mediante *espera activa*, es decir, mediante bucles de consulta repetida de los valores de ciertas variables hasta que se compruebe que se cumplen las condiciones esperadas.

2.2. Esquemas de interacción

La colaboración entre varios procesos exige, en general, alguna forma de interacción entre ellos. Se han caracterizado algunos esquemas típicos de interacción, tales como los que se describen a continuación.

Un *esquema de cooperación* se caracteriza porque la información fluye de unos procesos a otros, al tiempo que cada uno va realizando su parte del trabajo. Es este esquema hay unos procesos *emisores* que envían información a otros procesos *receptores*. La información puede transferirse por medio de un *gestor* que realiza una abstracción de datos adecuada para el paso de la información. El ejemplo clásico de cooperación es el problema del productor y el consumidor.

En estos esquemas la sincronización se plantea básicamente como una *ordenación temporal*. Por ejemplo, el tratamiento de unos datos en el proceso receptor ha de ser necesariamente posterior a su elaboración en el proceso emisor.

Un *esquema de competencia* contiene un conjunto de procesos que pretenden usar ciertos *recursos compartidos*. Cuando varios procesos *usuarios* compiten por el uso de un mismo recurso pueden ser coordinados mediante un *gestor* que planifica el acceso y asegura a los usuarios que dispondrán de él en las condiciones deseadas. Ejemplos clásicos de competencia son el problema de la "cena de los filósofos" y el problema de los lectores y los escritores.

En estos esquemas la sincronización se plantea básicamente como un problema de *exclusión mutua*, de manera que se eviten determinadas combinaciones en el acceso simultáneo al recurso.

El acceso a recursos compartidos se plantea generalmente en forma de *acciones atómicas*. Una acción atómica es aquella que se desarrolla totalmente en un proceso sin interferencia por parte de otros procesos. Las acciones atómicas tienen la propiedad de *serializabilidad*: el resultado global de invocar varias operaciones atómicas de forma simultánea desde diferentes procesos es equivalente a la realización sucesiva de dichas operaciones, una tras otra, en algún orden.

2.3. Propiedades de los programas concurrentes

Como cualquier otro programa, diremos que un programa concurrente es correcto si satisface su especificación. Tradicionalmente esa especificación se desglosa en dos partes, denominadas propiedades de seguridad y propiedades de vivacidad.

Una *propiedad de seguridad* especifica una condición que debe cumplirse en todo momento durante la ejecución del programa. Frecuentemente declara una restricción, es decir, *un conjunto de estados que no deben ser alcanzados* por el programa concurrente. Así, una propiedad de seguridad de un programa que sincroniza el acceso de varios procesos a una cuenta corriente compartida puede ser que el saldo no se haga negativo; de un programa que gestiona el acceso a un aparcamiento, que no haya en un momento dado más coches que la capacidad, etc.

Una *propiedad de vivacidad* especifica una condición que debe ocurrir eventualmente durante la ejecución del programa. A menudo se enuncian como *en algún momento se entra en un estado deseable o se produce un dato necesario*. Estas propiedades se suelen considerar una vez que se han asegurado las propiedades de seguridad.

Algunas de estas propiedades son: situaciones en la que uno o más procesos no pueden progresar (*interbloqueos*[2]), situaciones en las que no se garantiza la progresión de un proceso (*inanición*), la *equidad* – posibilidades similares de ejecución para procesos en competencia –, etc.

Hay veces en las que la especificación de un problema *fuerza* a ciertos comportamientos de los procesos que pueden ir en contra de los principios que acabamos de enunciar. Llamamos a estas restricciones *propiedades de prioridad*. Una propiedad de prioridad especifica, en caso de competencia entre procesos, su prioridad relativa.

Por comodidad, llamaremos *sincronización de seguridad* y *sincronización de vivacidad/prioridad* a la parte del código que garantiza las propiedades respectivas.

[2]En general, la ausencia de interbloqueos es formalmente una propiedad de seguridad, sin embargo, en el contexto de este documento aparecerá específicamente enunciada como propiedad de vivacidad por ser lo más coherente con las fases en que se estructura la metodología.

2.4. Mecanismos de sincronización

A lo largo de la historia de la programación concurrente se han ido proponiendo diversos mecanismos para la descripción y garantía de la exclusión mutua y la sincronización condicional [AS89]. Se puede encontrar una descripción detallada de los mecanismos de sincronización que veremos en esta sección en [And91].

2.4.1. Espera activa

El análisis de los algoritmos iniciales que aseguran exclusión mutua utilizando como única herramienta variables y bucles de espera puede ayudar a entender, a un escala pequeña, los problemas a los que debe hacer frente la programación concurrente. En [BA82] y [And91] pueden encontrarse varios ejemplos.

El uso de estos algoritmos está, en general, absolutamente desaconsejado[3] por lo que inmediatamente pasaremos a describir mecanismos con un mayor nivel de abstracción.

2.4.2. Semáforos

Los semáforos (ver [BA82, capítulo 4]) deben considerarse como tipos abstractos de datos cuyas operaciones principales (`Wait` y `Signal`) se realizan en exclusión mutua e incluyen la suspensión del proceso que las ejecuta bajo ciertas condiciones. Adicionalmente existe una operación de inicialización a llamar una única antes de cualquier otro acceso al semáforo. Una posible especificación del funcionamiento de un semáforo es:

sem $\in \mathcal{N}$

`Init(sem, n)` $\equiv \langle$ `sem := n` $\rangle$
`Wait(sem)` $\equiv \langle$ **AWAIT** `sem` $> 0 \rightarrow$ `sem := sem-1` $\rangle$
`Signal(sem)` $\equiv \langle$ `sem := sem + 1` $\rangle$

Un semáforo inicializado a cero garantiza la ejecución exclusiva de una serie de instrucciones mediante el siguiente protocolo

```
Wait (semáforo);
usar recurso
Signal (semáforo);
```

basado en la asociación de un semáforo a cada recurso diferenciado. Si desde otro proceso se intenta ejecutar `Wait` mientras el primero usa el recurso, el segundo quedará bloqueado hasta que el primero ejecute `Signal`.

El mecanismo de semáforos se considera de bajo nivel y poco estructurado, ya que el fragmento de programa que ha de realizarse de forma exclusiva (llamado *sección crítica*) no aparece estructurado de forma diferenciada dentro del programa que lo incluye, y porque al estar repartido por fragmentos "distantes" de código los diferentes usos de cada semáforo, la localización de errores – e.g. descubrir que se ha producido un interbloqueo por ejecutar una secuencia de `Wait` y `Signal` en el orden equivocado – suele ser una ardua tarea.

2.4.3. Regiones críticas

Para obviar el inconveniente anterior se propuso el esquema de *región crítica*, como estructura de programa que realiza directamente una sección crítica:

```
region recurso do
    usar recurso
end region
```

[3] Aunque pueden tener utilidad en dispositivos muy específicos donde hay procesadores completamente dedicados a una única tarea y que, por tanto, no desperdician recursos ejecutando un bucle vacío.

Ahora el recurso se designa directamente por su nombre, y la sección crítica aparece bien estructurada. La forma básica de región crítica resulta insuficiente para ciertos problemas, en que el acceso (exclusivo) al recurso tiene además ciertas restricciones, de manera que determinadas operaciones con el recurso sólo pueden realizarse si éste se encuentra en un estado apropiado. Si no es así, hay que esperar hasta que se alcance dicho estado, lo cual ocurrirá como consecuencia de las operaciones realizadas con el recurso por parte de otros procesos.

2.4.4. Regiones críticas condicionales

El mecanismo de región crítica ampliado con una primitiva de espera por condición se denomina *región crítica condicional*. El esquema de la sección crítica suele ser:

```
region recurso do
    preparación
    await condición necesaria;
    usar recurso
end region
```

Obsérvese que no habría bastado con usar regiones críticas más expresiones condicionales, ya que o no garantizaríamos seguridad o caeríamos en los problemas de la espera activa.

Al implementar regiones críticas condicionales, el sistema en tiempo de ejecución es el encargado del desbloqueo de los procesos cuando las condiciones por las que esperan pasan a ser ciertas. Ésto puede ser una fuente de ineficiencia, pues el cambio de cualquier variable durante la ejecución del programa es susceptible de desbloquear procesos que estuviesen suspendidos.

2.4.5. Monitores

La programación del acceso al recurso compartido puede estructurarse de forma mejorada agrupando en una misma unidad de programa todo el código de las operaciones de acceso. El mecanismo de monitores [Hoa74], en concreto, consigue dicho efecto.

Podemos ver un monitor como un tipo definido siguiendo el siguiente esquema:

```
type Tipo_del_Monitor = monitor
    use elementos externos;
    public Operación1, ..., OperaciónM;
    state variables locales, protegidas;
    Q1, ..., QC: condition;
    procedure Operación1 (parámetros);
    ...
    procedure OperaciónM (parámetros);
    ...
begin
    inicialización
end monitor;
```

Una vez definido el tipo monitor, se pueden ya declarar variables de ese tipo:

Nombre_del_monitor : *Tipo_del_monitor*;

En esta sección, las operaciones sobre el monitor se invocan usualmente de forma cualificada, con la notación de punto usual para referenciar elementos de un registro o métodos en el paradigma de orientación a objetos:[4]

[4] Esto, en el fondo, depende de decisiones de la sintaxis del lenguaje de programación que incluya monitores, pero la mayor parte de los que lo hacen se adhieren al paradigma de la programación orientada a objetos, con lo que esta notación es la más natural.

Nombre_del_Monitor.*OperaciónX* (*parámetros*);

La idea general del monitor es que las variables locales de estado (`state`) constituyan el contenido del recurso compartido, que queda protegido por el hecho de que sólo se puede acceder a dichas variables invocando alguno de los procedimientos públicos.

Sin embargo, la sincronización por condición ha de ser programada de forma explícita, mediante primitivas de bloqueo y desbloqueo que podemos ver como de más bajo nivel que el sencillo `await` de las regiones críticas condicionales. El tipo `condition` se usa para referirse a colas de procesos en espera dentro de un monitor.

Las únicas operaciones permitidas sobre una variable *c* de tipo condition se denominan *Delay* y *Continue*, y su semántica en la propuesta original de Hoare es la siguiente:

`Delay` (*c*) = el proceso que la invoca se queda en espera en la cola *c*.
Se permite la entrada de otros procesos al monitor.
`Continue` (*c*) = si hay procesos esperando en *c*, entonces se "despierta" uno de ellos;
si no hay, no se hace nada

Así, cuando en el código de una operación del monitor se detecta que no se dan las condiciones necesarias para su completa ejecución, se ejecutará `Delay` en una variable condition con la esperanza de que otro proceso ejecute el `Continue` correspondiente cuando se detecte que el estado del monitor permite reanudar la operación suspendida.

A cambio de una programación más sofisticada que con regiones críticas condicionales, el programador obtiene un control muy fino de la ejecución de los procesos dentro del monitor, que permite resolver adecuadamente los problemas de vivacidad que se estudiarán más adelante, así como una mayor eficiencia. Mecanismos con gran similaridad a los de monitores aparecen en varios lenguajes, como Java y C#, y pueden simularse de forma bastante sencilla en Ada.

2.4.6. Objetos protegidos

Por último, el lenguaje Ada 95 propone los objetos protegidos (*protected types*) como una forma de retomar las virtudes de las regiones críticas condicionales – desbloqueo automático – manteniendo la encapsulación propia de los monitores.

La gran ventaja respecto a los monitores es que para la mayoría de los problemas la programación de la sincronización condicional se simplifica notablemente, reduciéndose también la posibilidad de error provocado por una mala secuencia de ejecución de `Continues`.

Para aquellos casos donde una operación de un tipo protegido debe ser suspendida tras haber comenzado a ejecutarse, Ada introduce la primitiva `requeue` que permite "aparcar" momentáneamente dicha operación. La diferencia con el `Delay` de los monitores está en que el desbloqueo de una operación aparcada es también automático, es decir, no hay necesidad de un análogo al `Continue`.

No obstante, como ya hemos mencionado anteriormente, hay problemas para los que el programador requiere, ya sea por eficiencia o por cuestiones de prioridad, un control al detalle de la sincronización condicional que sólo una programación explícita de desbloqueos puede darle. Para esos casos veremos técnicas que nos permitirán *simular* el comportamiento de un Continue dentro del mecanismo de objetos protegidos.

La programación en Ada con objetos protegidos se trata en detalle a partir de la sección 3.

2.5. Concepto de equidad

Los mecanismos básicos de sincronización implican esperas de procesos y su posterior reanudación. Si hay varios procesos esperando por la misma condición y se ordena la continuación de uno de ellos, ¿a cuál se ha de dar paso?

En general se exige que las primitivas de sincronización cumplan un requisito de *equidad*, que podría expresarse diciendo que si se cumple en reiteradas ocasiones la condición por la que espera

un proceso, éste debe poder reanudarse en un tiempo finito. Dicho de otra manera, un proceso que espera no debe ser postergado indefinidamente frente a otros en situación similar.

Una forma sencilla de garantizar la equidad es hacer que los procesos esperen en una cola *FIFO*, de manera que siempre se da oportunidad de continuar al que lleva más tiempo esperando. En realidad el concepto de equidad es menos restrictivo y no exige el tratamiento en un orden estricto. Bastaría con que al extraer repetidamente un elemento de esa cola cualquier proceso acabara siendo elegido en un número finito de extracciones.

3. Objetos protegidos (*Protected types*)

Ada 95 permite declarar familias de recursos mediante *tipos protegidos*, que encapsulan, al igual que los monitores, un estado privado y una serie de operaciones que se han de realizar en exclusión mutua. El adjetivo "protegido" se refiere fundamentalmente a esta propiedad. Posteriormente podremos declarar recursos individuales como instancias (es decir, variables) de ese tipo (y cada una de las variables será un *objeto protegido*).

3.1. Declaración del tipo protegido

En primer lugar se debe declarar el tipo[5] con el que modelamos la familia de recursos:

```
protected type Tipo_Protegido is
    entry Operación1 (parámetros);
    ...
    entry OperaciónN (parámetros);
private
    declaración de variables y constantes
    entry Operación_Aplazada1 (parámetros);
    ...
    entry Operación_AplazadaM (parámetros);
end Tipo_Protegido;
```

Las operaciones públicas de un tipo protegido reciben el nombre especial de *entradas* (*entries*). Las entradas aplazadas son privadas y sólo pueden ser invocadas desde dentro del código de otras entradas del mismo recurso como una forma de partir en dos fases la ejecución de una operación pública que queda momentáneamente suspendida.

El código de cada operación se define aparte, en el *cuerpo* correspondiente:

```
protected body Tipo_Protegido is
    operaciones auxiliares

    entry Operación1 (parámetros)
    when condición is
    begin
        ...
    end Operación1;
    ...
    entry OperaciónN (parámetros)
    when condición is
    begin
        ...
```

[5] Describimos aquí los esquemas de definición más habituales. Para una descripción más detallada, como una gramática o aspectos peculiares de los objetos protegidos como la declaración de funciones y procedimientos públicos, el lector puede acudir a textos como [Coh95, BW98] o directamente al manual de referencia del lenguaje [TDBE01]

```
    end OperaciónN;
    entry Operación_Aplazada1 (parámetros)
    when condición is
    begin
        ...
    end Operación_Aplazada1;
    ...
    entry Operación_AplazadaM (parámetros)
    when condición is
    begin
        ...
    end Operación_AplazadaM;
end Tipo_Protegido;
```

Una vez definido el tipo, se pueden ya declarar variables de ese tipo:

```
Objeto_Protegido : Tipo_Protegido;
```

Las operaciones sobre el objeto protegido se invocan de forma cualificada, con la notación de punto usual para referenciar elementos de un registro:

```
Objeto_Protegido.OperacionX (parámetros);
```

Las condiciones de las cláusulas `when` en el cuerpo de cada entrada sirven para especificar la sincronización por condición, de manera análoga al `await` de las regiones críticas condicionales aunque con una importante restricción: **no se puede hacer referencia a los parámetros de la *entry***.

La transferencia de control desde una operación pública a otra aplazada privada se realizará mediante una instrucción especial `requeue`, que soslaya el interbloqueo que causaríamos por la invocación de una operación en exclusión mutua desde dentro de una situación de exclusión mutua. Explicaremos el uso de esta instrucción en la sección 3.3.

3.2. Tipos protegidos vs. tipos abstractos de datos

Los tipos protegidos tienen muchas similitudes con los tipos abstractos de datos, si bien no es conveniente llevar la analogía demasiado lejos, pues en estos últimos la noción de estado es ajena o secundaria, mientras que en el contexto de la concurrencia el estado del recurso es la característica más interesante desde el punto de vista del analista o diseñador. En esta sección se mostrarán las principales características de los objetos protegidos de forma bastante abstracta, suficiente para nuestros objetivos, en [BW98] puede encontrarse la semántica exácta de los mismos.

Dejando a un lado estas diferencias semánticas, es indudable que podemos utilizar la similitud formal entre recursos compartidos y tipos abstractos de datos por ejemplo, para usar una notación similar a la de éstos para especificar su comportamiento. Si bien la característica distintiva de los tipos protegidos es la exclusión mutua entre sus operaciones, ello no afecta a su especificación, ya que no introduce nada nuevo respecto a la situación de secuencialidad que se asume en las especificaciones de tipos abstractos de datos.

Como ya se ha dicho, para cada objeto instancia de un tipo protegido sólo una de las entradas puede estar en ejecución en un momento dado. Si un proceso invoca uno de esos procedimientos mientras otro se está ejecutando todavía como consecuencia de una invocación anterior por parte de otro proceso, el nuevo proceso quedará en espera hasta que haya terminado la operación en curso.

Si varios procesos intentan a la vez usar un objeto protegido, sólo uno de ellos conseguirá el acceso inicial, y los demás quedarán en espera hasta que les toque el turno a medida que van

terminando de usar el objeto uno tras otro.[6]

De esta manera, en ausencia de esperas programadas dentro del objeto protegido – e.g. mediante el uso de *requeue* – las operaciones de acceso están totalmente serializadas. El código de las operaciones puede ser el mismo que para un dato abstracto usado en un programa puramente secuencial. El sistema en tiempo de ejecución se encarga de forma automática y transparente de gestionar la contención necesaria cuando el objeto se usa de forma concurrente.

El objeto implementa directamente, por lo tanto, la especificación de un recurso compartido abstracto con el esquema siguiente:

C-TADSOL TipoRecurso
USA elementos externos

OPERACIONES
ACCIÓN Operación1: *TipoRecurso[es]* $\times$ *Tipo1[e]* $\times \cdots \times$ *TipoM[s]*
...
ACCIÓN OperaciónN: *TipoRecurso[es]* $\times$ *Tipo1[e]* $\times \cdots \times$ *TipoQ[s]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Recurso* = ...
DONDE: Tipo_Adicional = ...

INVARIANTE: ...

INICIAL(r**):** *Fórmula que especifica el valor inicial del recurso*

PROTOCOLOS: ...
CONCURRENCIA: *Ninguna – todas las operaciones son excluyentes*

La exclusión total se impone sólo entre las acciones simples sobre el objeto protegido, de manera que se pueden tener también operaciones compuestas con concurrencia aparente ejecutadas entrelazando operaciones simples. La cláusula de *protocolos* se encarga de reflejar esto:

PROTOCOLOS: ProtocoloA : ... – *Combinación de Operación1 ... OperaciónM*
...
ProtocoloK : ... – *Combinación de Operación1 ... OperaciónM*

CONCURRENCIA: ...

El significado es el siguiente: cada protocolo representa las *únicas* secuencias de llamada a entradas del objeto desde un proceso dado. Se asume que cada uno de estos protocolos puede entrelazarse, gozando, por tanto, de concurrencia aparente.

La cláusula de protocolos no tiene una correspondencia directa con ninguna estructura de programa en la declaración o cuerpo del tipo protegido, sino que es una indicación al programador que puede, o no, afectar a la programación del recurso o las tareas que acceden a él.

[6] La política que el lenguaje Ada impone para estos turnos no es trivial: distingue entre el turno para las llamadas que han quedado bloqueadas en el *exterior* del objeto protegido, las que han quedado bloqueadas en el *interior* del objeto protegido (tras hacer un *requeue*) y la selección, en ambos casos, entre el rearranque de tareas en el caso en que las guardas de varias operaciones se hagan ciertas simultáneamente. Dado que los diferentes lenguajes y mecanismos de concurrencia pueden definir e implementar distintas políticas, es más seguro suponer, a este nivel, la ausencia casi completa de cualquier comportamiento favorable, que en cualquier caso sólo afectaría cuestiones de vivacidad de la solución, y retrasar el afinamiento de ese comportamiento a estados posteriores de la implementación.

3.3. Objetos protegidos y sincronización por condición

Los objetos protegidos proporcionan mecanismos de alto nivel para realizar *sincronización por condición*. Las entradas de un tipo protegido pueden estar condicionadas mediante una *guarda*:

```
protected body Tipo_Protegido is
    ...
    entry OperaciónX (parámetros)
    when condición is
    begin
        ...
    end OperaciónX;
    ...
end Tipo_Protegido;
```

Si en el momento de llamar un proceso a la entrada *OperaciónX* no se cumple *condición*, dicho proceso se quedará bloqueado hasta que se cumpla y la exclusión mutua del objeto protegido permita la ejecución.

Es importante señalar que la condición sólo puede referirse a *variables de estado del objeto protegido*, nunca a los parámetros de entrada de la operación. Esto es comprensible si asumimos que la evaluación de las guardas es previa a la decisión indeterminista por alguno de los procesos en espera en alguna de las colas. Esto se justifica, en parte, por motivos de eficiencia: si las condiciones para aceptar una llamada pudiesen depender de algunos de sus argumentos, el número de guardas a evaluar en un momento dado no sería fijo, sino que dependería del número de procesos bloqueados en las diferentes guardas de entrada al objeto protegido.

Por tanto, cuando un proceso termina de ejecutar una operación del objeto, se evalúan las guardas de todas las operaciones y, si hay algún proceso en la cola de espera de una entrada, se selecciona para su ejecución, y si no, no se hace nada. Si, por el contrario, un proceso invoca una entrada de un objeto protegido cuando ningún otro proceso está ejecutando otra entrada y no hay procesos en espera en ninguna de las colas de entrada, ejecutará directamente la llamada si la guarda – que habrá sido evaluada con anterioridad – es cierta, y quedará bloqueado en espera en caso contrario (ver [BW98, Sección 7.4]).

¿Qué hacer si una precondición de una operación del recurso que estamos intentando modelar depende de los datos de entrada? Aquí es donde entran en juego las *entradas aplazadas*. La entrada pública tendrá una guarda no necesariamente igual a la deseada, teniendo sólo en cuenta las variables de estado del objeto protegido pero, una vez dentro de ella, se podrán usar los parámetros de entrada para evaluar la precondición completa. Si ésta es cierta, la operación se podrá ejecutar sin riesgo.

Pero, ¿qué ocurre si la guarda se evalúa como falsa? En tal caso podremos *copiar* los parámetros de entrada de la operación a variables locales del objeto protegido y *transferir el control* a una entrada privada del objeto. Al haber sido copiados los parámetros al estado del objeto, esta segunda entrada sí puede codificar completamente su CPRE en la guarda:

```
protected body Tipo_Protegido is
    ...
    entry OperaciónX (parámetros)
    when condición is
    begin
        ...
        if not condición completa
        then
            requeue Operación_PrivadaY;
        end if;
        ...
    end OperaciónX;
```

```
    ...
    entry Operación_PrivadaY (parámetros)
    when condición completa is
    begin
        ...
    end Operación_PrivadaY;
end Tipo_Protegido;
```

Se trata, pues, de dividir en dos aquellas operaciones cuya CPRE depende de los datos de entrada. Cuando decimos que la primera operación transfiere el control a la segunda, nos estamos refiriendo a que, al terminar ésta no se retorna al punto inmediatamente posterior al requeue. Esto explica el condicional en el esquema.

Veamos con más detalle los esquemas de código que pueden aparecer a la hora de implementar un recurso compartido mediante objetos protegidos. La sección 3.4 muestra diferentes esquemas que pueden darse según dependan o no CPREs de parámetros de entrada, número de procesos que llaman a una operación, etc. La sección 3.5 trata la cuestión de los problemas de exclusión parcial. Finalmente, la sección 3.6 desarrolla técnicas que permiten un control más fino sobre la secuencia de desbloqueo, algo que resultará útil a la hora de afinar las propiedades de vivacidad de un objeto protegido.

3.4. Esquemas de código

Con lo visto hasta ahora, a cada operación pública de un recurso compartido le corresponde una entrada pública en el objeto protegido y, quizás, una o más entradas privadas en caso de que su precondición dependa de los datos de entrada:

```
protected body Tipo_Protegido is
    declaración de variables y constantes
    entry Operación1 (parámetros)
    when condición is
    begin
        ...
    end Operación1;
    ...
    entry OperaciónN (parámetros)
    when condición is
    begin
        ...
    end OperaciónN;
    entry Operación_Aplazada1 (parámetros)
    when condición is
    begin
        ...
    end Operación_Aplazada1;
    ...
    entry Operación_AplazadaM (parámetros)
    when condición is
    begin
        ...
    end Operación_AplazadaM;
end Tipo_Protegido;
```

Pasemos a especificar con más detalle en qué circunstancias se podrán usar esquemas más o menos sofisticados:

3.4.1. Sincronización independiente de los parámetros de entrada

Cuando la CPRE no depende de los datos de entrada, la traducción es bastante inmediata:

```
entry OperaciónX (parámetros)
-- su CPRE no depende de los parámetros de entrada
when CPRE is
begin
   -- aquí se cumple CPRE
    implementación de la Operación X
   -- aquí se debe cumplir POST
    código de depuración de POST
end OperaciónX;
```

Obsérvese que estamos utilizando al máximo la información de especificación del recurso: la de la CPRE en la guarda y la de la POST al final del código de la operación como código de depuración que debería lanzar una excepción en caso de evaluar a Falso. Para ello se pueden utilizar pragmas de depuración de Ada95: `pragma Assert(`*condición*`)`, por ejemplo.

3.4.2. Sincronización dependiente de datos de entrada

La implementación se complica cuando la CPRE depende de los parámetros de entrada. La traducción directa estaría prohibida por el compilador, lo que nos obligará a crear una entrada pública y una o más entradas privadas haciendo uso de la posibilidad de *reencolado* de Ada (`requeue`). Vamos a analizar distintas posibilidades mostrando en cada caso sus pros y sus contras.

Indexación de casos

La cuestión fundamental es: ¿cuántos casos distintos existen debido a esos datos de entrada involucrados en la CPRE? En el peor de los casos tendremos tantas entradas privadas como versiones de la CPRE podamos obtener instanciando los parámetros que la afectan. Formalmente, una CPRE Φ que depende de un parámetro a sobre un dominio D tiene tantas versiones como el cardinal de D.

Si $D = \{x_1, \ldots, x_n\}$ es un dominio finito, Φ se puede desglosar en las instancias[7]

$$\Phi[a := x_1]$$
$$\vdots$$
$$\Phi[a := x_n]$$

La implementación más evidente pasaría por abrir la operación con una guarda `True` y comprobar cada caso para *reencolar* sobre una operación asociada para cada uno. El proceso se repetiría en caso de haber otros parámetros y el esquema resultante es similar al siguiente:

```
entry OperaciónX (a_1 ... a_n, b_1 ... b_m)
-- su CPRE depende de los a_i
when True is
begin
    if a_1 = A_1 ∧ ··· ∧ a_n = A_n then
        requeue OperaciónXA_1 ... A_n;
    elsif
        ...
    end if;
end OperaciónX;
```

[7] $P[y := e]$ representa la fórmula P después de substituir todas las apariciones de la variable y por la expresión e.

```
...
entry OperaciónXA1 ... An  (a1 ... an, b1 ... bm)
-- esta entrada ha sido declarada privada
when CPRE[a1 ↦ A1 ... an ↦ An] is
-- la CPRE se codifica en su totalidad
begin
    -- aquí se cumple la CPRE
     implementación de la Operación XA1 ... An
    -- aquí se debe cumplir POST
     código de depuración de POST
end OperaciónXA1 ... An;
```

Este es el esquema seguido en la solución al problema del *buffer de pares e impares* (Sec. 5.2).

Familia de entradas

La replicación manual de operaciones que se acaba de presentar no parece muy elegante, sobre todo si se tiene en cuenta que Ada95 permite ese tipo de réplicas automáticamente gracias a la posibilidad de definir una familia de entradas [BW98, Capítulos 7 y 8]. Puede verse un ejemplo de uso de la técnica de replicación de entradas en la solución al problema del *buffer de pares e impares* (Sec. 5.2).

Indexación de procesos

Puede que el número de elementos en el dominio D sea finito, pero si es excesivamente elevado las técnicas de replicación, manuales o automáticas, darían lugar a un número intolerable de operaciones en el recurso.[8] Una posibilidad para conseguir el mismo efecto sin replicación manual de código se basa en la observación de que el número de procesos en ejecución que pueden invocar una entrada determinada va a ser, a menudo, mucho menor que el resultante de instanciar su CPRE con todos los valores posibles de aquellos parámetros de que depende.

Procederemos entonces de la siguiente manera: en lugar de tener una entrada privada por cada posible CPRE, tendremos una entrada privada por cada posible proceso llamante. Una forma de conseguirlo es añadiendo un parámetro que identifique al proceso que la invoca, de tal manera que nunca vamos a tener más de un proceso esperando en cada *cola* interna. Aquí sí necesitaremos realizar una copia local de los parámetros con que se llama a la operación, más exactamente, tantas copias como procesos, copias almacenadas en *tablas* cuyas claves van a ser los identificadores de los procesos:

```
entry OperaciónX  (PID, a1 ... an, b1 ... bm)
-- su CPRE depende de los ai, pero sabemos que nunca tendremos que atender simultáneamente
-- 2 llamadas con el mismo valor de PID
when True is
begin
     -- copiamos los parámetros que afectan a la CPRE en vectores a'i
     a'1(PID):=a1;
     ...
     a'n(PID):=an;
     requeue OperaciónX(PID);
end OperaciónX;
...
entry OperaciónX(PID)  (a1 ... an, b1 ... bm)
-- esta entrada ha sido declarada privada
when CPRE[a1 ↦ a'1(PID) ... an ↦ a'n(PID)] is
```

[8] En casos extremos el programa se romperá nada más arrancar.

```
-- la CPRE se codifica en su totalidad
begin
   -- aquí se cumple la CPRE
    implementación de la Operación X(PID)
   -- aquí se debe cumplir POST
    código de depuración de POST
end OperaciónX(PID);
```

La forma más cómoda de realizar esta replicación es, de nuevo, mediante una familia de entradas. Éste, esencialmente, es el esquema que se ha usado en la primera solución al problema de la *cuenta bancaria compartida* (Sec. 5.4).

Hay casos en que la necesidad de respetar estrictamente un interfaz impide añadir identificadores de procesos a las operaciones públicas. En esta situación se puede utilizar un esquema algo más trabajoso, pero esencialmente similar, que asigne un identificador diferente a cada llamada, extraído de entre un conjunto finito de ellos, que forma parte del estado (aumentado) del objeto protegido. Esta extracción se realizaría como parte de las acciones entre la aceptación de la primera llamada del objeto protegido y el reencolado. Este identificador se utilizaría para realizar la indexación en la replicación de código. El código quedaría, por tanto, con un aspecto similar al siguiente:

```
entry Operación (a_1 ... a_n, b_1 ... b_m)
-- su CPRE depende de los a_i, pero sabemos que hay un límite superior al número de llamadas que
-- podemos que tener que atender simultáneamente
when True is -- Siempre admitimos una llamada
  I: Tipo_PID;
begin
    while Vacio(Tabla, I) loop -- Deberíamos hacerlo más eficiente
      I := I + 1; -- Debemos estar seguros de que hay sitio para todos los procesos llamantes
    end loop;
    -- copiamos los parámetros que afectan a la CPRE en la entrada correspondiente
    Asignar(Tabla, I, (a_1, a_2, ..., a_n));
    requeue OperaciónApl(I);
end Operación;
...
entry OperaciónApl(I) (a_1 ... a_n, b_1 ... b_m) -- Esta entrada ha sido declarada privada
when CPRE[(a_1, ..., a_n) ↦ Recuperar(Tabla, I)] is
  -- la CPRE se codifica en su totalidad utilizando los valores de llamada
  -- anteriormente guardados en la tabla
begin
   -- aquí se cumple la CPRE
    <implementación de la Operación(PID)>
    Borrar(Tabla, I); -- Liberar posición ya innecesaria
   -- aquí se debe cumplir POST
    <código de depuración de POST>
end OperaciónApl(PID);
```

Nótese que siempre aceptamos la llamada, y que suponemos que hay sitio en la tabla para almacenar el número máximo de invocaciones simultáneas que se puedan realizar a *Operación* en un momento dado. La manera más sencilla de asegurar que esto es así es reservar espacio para tantas entradas como procesos puedan efectuar esa llamada. Si ese número no se conoce *a priori* o es excesivamente grande, puede establecerse una limitación previa cambiando la guarda `when True` de *Operación* por `when N_Entradas_Llenas < Max` (con una actualización adecuada de `N_Entradas_Llenas`) o, usando directamente un interfaz razonable de tablas, por `when not Llena(Tabla)`. Pero eso sólo sería necesario si el número de posibles procesos llamadores simultáneamente es prohibitivamente alto, lo que no suele ser el caso común. En la peor de las

situaciones, imponer innecesariamente una restricción de ese tipo podría llevar a que el recurso se comportase de manera indeseable, bloqueándose al no admitir la entrada de una llamada cuya guarda podría evaluarse a *true* y posteriormente desbloquear otras llamadas suspendidas.

Atención a procesos de uno en uno

Continuando con el ejemplo anterior, en algunas ocasiones los requisitos del problema permiten aplicar otras técnicas, más o menos *ad hoc*, reducir el número de entradas replicadas y, al mismo tiempo, evitar el uso de tanto espacio para copias de parámetros. En un caso extremo (pero no por ello infrecuente) el problema puede permitir que se bloquee y atienda a los procesos de uno en uno. Así, hasta que el primero no es atendido no se consideraría la evaluación de las guardas de los demás. En esta situación es posible aplicar un esquema en el que una vez que hay un proceso esperando en la operación privada se impide la llamada a la pública. Llamamos a esto *atención de uno en uno* o *por orden de llegada*:

```
entry OperaciónX (a_1 ... a_n, b_1 ... b_m)
-- su CPRE depende de los a_i
when not BloqueadaX is
begin
    -- copiamos los parámetros que afectan a la CPRE
    a'_1 := a_1;
    ...
    a'_n := a_n;
    BloqueadaX := True;
    requeue OperaciónX';
end OperaciónX;
...
entry OperaciónX' (a_1 ... a_n, b_1 ... b_m)
-- OperaciónX' ha sido declarada privada
when CPRE[a_1 ↦ a'_1 ... a_n ↦ a'_n] is
-- CPRE se codifica en su totalidad gracias a la copia de los parámetros
begin
   -- aquí se cumple CPRE
    BloqueadaX := False;
    implementación de la Operación X
   -- aquí se debe cumplir POST
    código de depuración de POST
end OperaciónX';
```

Este esquema sólo necesita espacio para una copia de los parámetros.[9] Se ha usado en una de las soluciones al problema de la *cuenta bancaria compartida* (Sec. 5.4) en la que, para evitar inanición de peticiones elevadas, se atienden los reintegros por estricto orden de llegada.

Es muy importante observar que esta técnica no es aplicable en general puesto que reduce seriamente la concurrencia de un sistema y sólo está justificada su aplicación cuando los requisitos del problema lo permiten.

3.5. Gestores de sincronización

Hay situaciones en que el objeto protegido no puede desempeñar por sí mismo el papel del recurso compartido. Esto ocurre, en particular, cuando la exclusión mutua completa es una restricción demasiado fuerte, y la aplicación debe tolerar el acceso simultáneo al recurso desde varios procesos en determinadas condiciones, o bien necesita procesar nuevas peticiones mientras se está usando el recurso. Estos problemas se denominan, a veces, de *exclusión parcial*.

[9] Y, en realidad, es una especialización del esquema general que utiliza tablas para el caso en que el tamaño de la tabla es 1.

En estas situaciones el resulta muy complicado, si no imposible, implementar el recurso compartido únicamente como un objeto protegido. El código de las operaciones ha de ser externo a un objeto protegido que se limitará a exportar procedimientos públicos para implementar protocolos de de inicio y finalización de las acciones correspondientes al recurso compartido. Dicho objeto protegido gestionará la sincronización de inicio, para lo que necesitará mantener información suficiente para comprobar la CPRE de la operación real del recurso. La información de la POST que influya en las CPRE deberá manejarse de forma adecuada. Veamos un esquema general:

```
protected type Tipo_Protegido is
    entry IniciarOperaciónX (parámetros); -- casi siempre de entrada
    entry TerminarOperaciónX (parámetros); -- casi siempre de salida
    ...
private
...
end Tipo_Protegido;
protected body Tipo_Protegido is
    entry IniciarOperaciónX (parámetros)
    when condición is
    ...
    end IniciarOperaciónX;
    entry TerminarOperaciónX (parámetros)
    when True is
    ...
    end TerminarOperaciónX;
...
end Tipo_Protegido;
```

Por supuesto, puede suceder que *IniciarOperaciónX* tenga que aplazarse con una entrada privada, si su CPRE depende de parámetros de entrada. Observa que la entrada *TerminarOperaciónX* tiene su guarda a *true*.

Los procesos usarán el objeto según el esquema:

```
Objeto_Protegido : Tipo_Protegido;
...
Objeto_Protegido.IniciarOperaciónX ...;
código de la operación
Objeto_Protegido.TerminarOperaciónX ...;
```

Este esquema recuerda claramente al protocolo de acceso exclusivo a un recurso mediante el mecanismo de semáforo. En la sección 2.4 se comentó que esta construcción presenta cierta debilidad estructural, debido a que en cada acceso al recurso el código del proceso cliente debe seguir explícitamente la disciplina indicada. En el caso de un verdadero esquema de competencia por un recurso, es posible mejorar la estructura a base de englobar el objeto en un módulo que implemente la abstracción de datos del recurso, de manera que la disciplina de llamadas está encapsulada en dicho módulo y los procesos clientes sólo invocan una operación global, de la forma:

```
package Recurso is
    procedure OperaciónX... ;
    ...
    variables del recurso;
    private
    protected type TipoGestor is
        entry IniciarOperaciónX... ;
        entry TerminarOperaciónX... ;
        ...
```

```
    end TipoGestor;
    ...
end Recurso;

package body Recurso is
Gestor : TipoGestor; -- gestor de sincronización
...
procedure OperaciónX... is
begin
    Gestor.IniciarOperaciónX;
    código de la operacion -- ¡OJO! código reentrante
    Gestor.TerminarOperaciónX
end OperaciónX;
...
end Recurso;
```

Con esta implementación los procesos clientes pueden invocar directamente:

```
Recurso.OperaciónX ...;
```

La solución para encapsular el recurso y el gestor de sincronización que se acaba de presentar no es la más elegante de todas. Un esquema más flexible, más elegante y más potente consiste en introducir un tipo abstracto de datos (`private` al menos) que encapsule toda la información asociada al recurso y al gestor de sincronización.

El módulo de control del recurso así organizado constituye, de alguna manera, un verdadero objeto protegido multi-hilo (*multi-threaded*). Si las condiciones lo permiten, varios procesos pueden estar ejecutando a la vez el código de la operación.

Hay que tener en cuenta que un fragmento de código (como el de los procedimientos anteriores) que pueda estar ejecutándose desde varios procesos a la vez debe ser *reentrante*. Para ello dicho código sólo debe hacer referencia a variables comunes a las diversas líneas de ejecución para las que se ha comprobado de antemano (en el gestor) que no habrá interferencia en su uso. Todas las variables auxiliares deberán ser locales al procedimiento que implementa la operación, lo que garantiza que existirán copias separadas para cada línea de ejecución.

Las precauciones anteriores han de extenderse al uso indirecto de variables definidas en las librerías del sistema (*system libraries*) o asociadas al compilador (*run-time libraries*). Esta variables pueden estar fuera del conocimiento del programador. La documentación de las librerías debería avisar de su existencia o ausencia, indicando si el uso de la librería es seguro en un ambiente "multi-thread" (*MT-safe*).

El paso de refinamiento descrito en este apartado puede describirse usando la notación para especificación de recursos, es decir, se puede hacer en dos fases: en la primera se pasa de una especificación con cláusula de concurrencia a otra sin ella, y en la segunda se traduce ésta al código de objetos compartidos.

3.6. Desbloqueo explícito

Como ya hemos comentado, hay situaciones en las que al programador le viene bien controlar la secuencia de desbloqueos que se desencadena cuando un proceso termina de ejecutar una operación del objeto protegido. Ello puede ser debido a restricciones de prioridad especificadas en el análisis del problema, para evitar problemas de vivacidad como la *inanición*, por motivos de eficiencia, etc.

Hay mecanismos de expresión de la concurrencia que giran alrededor de primitivas de bloqueo y desbloqueo explícito como los semáforos (`Signal`) o los monitores (`Continue`). Sin embargo, las regiones críticas condicionales y los objetos protegidos pertenecen a la clase de mecanismos en los que el desbloqueo es *ímplicito*, es decir, se dispara automáticamente al cumplirse cierta propiedad

del estado del recurso – en parte precisamente para evitar que quede al arbitrio de un programador poco cuidadoso.

El desbloqueo explícito con objetos protegidos es algo, pues, que ha de ser *simulado*. Hay dos formas relativamente sencillas de hacer esto. La primera consiste en bloquear a los procesos *fuera del objeto que encapsula la sincronización del recurso*, en otros objetos protegidos que vienen a interpretar un papel similar al de semáforos. La segunda consiste en codificar el desbloqueo explícito en la propia lógica de las guardas de las entradas del objeto protegido.

Desarrollaremos en lo que sigue esta segunda técnica. Se trata, esencialmente, de usar variables auxiliares – que llamaremos *de condición* por analogía con los monitores – para forzar ciertas guardas a Cierto o Falso, consiguiendo así despertar a los procesos deseados.[10]

Supongamos que tenemos una serie de procesos que han de bloquearse en una de entre varias condiciones que llamaremos *cond*1 ... *condN*, cada una asociada con una situación diferente, como veremos en la metodología de refinamiento progresivo de la próxima sección. Tendríamos el siguiente esquema para el tipo protegido:

```
type Condicion is (Ninguno, Cond1, ..., CondN);
protected type TipoRecurso is
    entradas públicas: se encargan de dar valor a Siguiente
private
    OpCond1 (parámetros);
    ...
    OpCondN (parámetros);
    ...
    Siguiente : Condicion := Ninguno;
end TipoRecurso;
protected body TipoRecurso is
    ...
    entry OpCond1 (parámetros)
    when Siguiente = Cond1 is
    begin
        Siguiente = Ninguno;
        ...
    end OpCond1;
    ...
    entry OpCondN (parámetros)
    when Siguiente = CondN is
    begin
        Siguiente = Ninguno;
        ...
    end OpCondN;
end TipoRecurso;
```

El desbloqueo explícito se produce, obviamente, al ejecutar un proceso

```
Siguiente := CondX;
```

inmediatamente antes de salir del objeto protegido. Este esquema es muy general y en la práctica se usan versiones especializadas (ver Sec. 4.3.4).

El esquema de desbloqueo explícito se puede combinar con los esquemas que hemos visto anteriormente. En las siguientes secciones se discute cómo integrar los diferentes esquemas y ejemplos de su aplicación.

[10] La técnica es incluso más potente que el *Continue* de monitores, pues permite simular la situación de un proceso esperando simultáneamente en varias colas.

4. Refinamiento progresivo

Al analizar las características deseables en una estructura de datos de acceso concurrente en memoria compartida, podemos distinguir dos aspectos metodológicos que pueden desarrollarse de forma relativamente independiente:

Abstracción de datos. La construcción debe realizar una abstracción de datos adecuada para la comunicación de información entre los procesos. Como en cualquier diseño modular, la definición de la abstracción debe tener sentido, con independencia de que su uso sea secuencial o concurrente.

Sincronización. Dado que los datos son compartidos por varios procesos, se debe sincronizar su uso. La sincronización debe garantizar las propiedades de seguridad y las propiedades de vivacidad/prioridad.

El mecanismo de objetos protegidos cubre ambos aspectos de una manera separada y elegante, pero acabamos de ver que en ciertos casos es necesario programar los desbloqueos pervirtiendo los mecanismos que el lenguaje proporciona, por lo que existe un riesgo de mezclar en una misma parte del código aspectos diferentes.

La presente propuesta sugiere separar los aspectos indicados durante el proceso de refinamiento del objeto compartido, de acuerdo con la siguiente secuencia:

1. Desarrollo de la abstracción de datos.
2. Desarrollo de la sincronización de seguridad.
3. Desarrollo de la sincronización de vivacidad/prioridad.

Este orden obedece al reconocimiento de que en la mayoría de los casos cada aspecto indicado ha de desarrollarse atendiendo a los anteriores.

4.1. Desarrollo de la abstracción de datos

Como se mencionó en la secciones 3.2 y 3.5, un objeto protegido puede implementar por sí mismo una abstracción de datos con acceso concurrente, o bien limitarse a gestionar funciones de sincronización. En el primer caso el desarrollo de la abstracción puede seguir los mismos pasos que una abstracción de datos sin problemas de concurrencia:

1. Definición de la abstracción de datos. Consistirá en identificar los atributos (contenido de información) de la abstracción, y la interfaz (cabecera, especificando los argumentos) de cada uno de los procedimientos públicos.
2. Implementación de la abstracción de datos. Consistirá en elegir las estructuras de datos internas adecuadas para representar los atributos, y escribir el código de los procedimientos públicos, y la inicialización de la abstracción. Al mismo tiempo se podrán desarrollar elementos locales auxiliares como resultado del refinamiento de la abstracción.

Si el objeto protegido actúa como gestor de una abstracción de datos externa, dicha abstracción de datos se desarrollará previamente de la forma indicada, usando los esquemas ya mencionados en la sección 3.5, iniciando y terminando el código de cada operacion con llamadas a los procedimientos apropiados del gestor.

Operación	CPRE informal	¿Dependiente de los parámetros?	CPRE codificada

Cuadro 1: Tabla de sincronización (bloqueo).

4.2. Desarrollo de la sincronización de seguridad

Comenzamos el desarrollo de la sincronización con el refinamiento de la sincronización de seguridad. Conviene resaltar que en esta etapa inicial *no se tienen en cuenta*, de momento, las propiedades de vivacidad/prioridad. Con ello se sigue la recomendación general de desarrollar mediante refinamientos progresivos, ignorando los detalles hasta que sea necesario tenerlos en cuenta.

Esta etapa del desarrollo puede realizarse también en dos pasos, separando la parte de especificación de la parte de implementación. En el primer paso se identifican los elementos de sincronización necesarios para garantizar las propiedades de seguridad. En el segundo paso se implementa dicha sincronización mediante el desarrollo del código correspondiente.

4.2.1. Diseño de la sincronización de seguridad

Para empezar, conviene recordar que en la mayoría de los casos las propiedades de seguridad corresponden a condiciones de consistencia entre el estado de los datos y la operación que se desea realizar; es decir, son *precondiciones* (condiciones de sincronización incluidas) de las operaciones de la abstracción.

De acuerdo con lo anterior, en este paso de diseño se deben identificar las restricciones que impiden abordar en condiciones normales una determinada operación, y que se corresponderán con guardas de entradas del tipo protegido. Si dichas restricciones no dependen de parámetros de entrada de las operaciones del recurso compartido, se codificarán directamente en la guarda de una entrada pública. En caso contrario, habrá una guarda a Cierto en la entrada pública y (al menos) una segunda entrada privada.

Estas situaciones de bloqueo pueden recogerse en una tabla con el formato que se muestra en la tabla 1. En esta fase no se considera la posibilidad de programar desbloqueos explícitamente.

Cada situación de bloqueo se describirá en una línea de la tabla, con el siguiente contenido de información:

Operación: Declara los procedimientos donde se produce el bloqueo, con sus argumentos formales – necesario, pues pueden intervenir en las precondiciones. Sí obviaremos a veces – principalmente por motivos de espacio – referirnos al parámetro de la especificación formal correspondiente *al propio objeto protegido*.

CPRE informal: Expresa informalmente la precondición de sincronización; puede expresarse de la forma más cómoda, es decir, en lenguaje natural, notación matemática, etc. Se expresa en términos del estado de la abstracción de datos, parámetros del procedimiento y estado de sincronización del objeto.

Dependencia de los parámetros de entrada: Se declara aquí si la CPRE de la operación en cuestión depende de algún parámetro de entrada.

CPRE codificada: Expresa la precondición completa, ya sea codificada en Ada 95 u otra notación suficientemente formal, usando las variables y funciones necesarias.

Como ejemplo, se presenta la tabla de precondiciones de sincronización correspondiente a un buffer con capacidad para MAX elementos, que en un momento dado tiene n elementos. Las operaciones son *Poner* y *Tomar* elemento. El resultado se muestra en la tabla 2.

La tabla debe prepararse y conservarse como un elemento de documentación independiente e incluirse parte de su contenido como comentarios o asertos de depuración en el código fuente.

Operación	CPRE informal	dep.	CPRE codificada
Poner (Item : TDato[ent])	Buffer no lleno	no	$n < MAX$
Tomar (Item : TDato[sal])	Buffer no vacío	no	$n > 0$

Cuadro 2: Tabla de bloqueos para un buffer acotado compartido.

Operación	CPRE informal	¿Dep.?	CPRE codificada
Op (...)	CPRE	sí	$\Phi[a \mapsto x_1]$
			$\Phi[a \mapsto x_2]$
			$\vdots$
			$\Phi[a \mapsto x_n]$

Cuadro 3: Tabla de bloqueos en la que se ha desglosado una CPRE dependiente de parámetros de entrada.

4.2.2. Implementación de la sincronización de seguridad

En este paso se integra el diseño anterior en el código del objeto protegido resultante de la implementación de la abstracción de datos. Ya hemos visto que esto es inmediato cuando las precondiciones no dependen de parámetros de entrada de la operación y casi inmediato cuando sí dependen, como se vio en la sección 3.4.

Si ha habido que desglosar una CPRE Φ que depende de un parámetro a en el dominio D (donde $D = \{x_1, \ldots, x_n\}$) en las instancias $\Phi[a := x_1] \ldots \Phi[a := x_n]$, se puede reflejar en la tabla de bloqueos tal como se muestra en la tabla 3. Por supuesto, cada una de las CPREs codificadas da origen a una entrada privada en el objeto protegido[11]. Hay que incluir código que decida, dependiendo del valor de dicho parámetro, qué *requeue* ejecutar.

Si el parámetro en cuestión pertenece a un dominio más complejo o con un alto número de elementos, el desglose de la CPRE habrá de realizarse con un nivel de detalle menor, o con técnicas *ad hoc*, a juicio del diseñador, tal como se puede ver en las secciones 3.4.2, 3.4.2, o en el ejemplo de la sección 5.4.

4.3. Desarrollo de la sincronización de vivacidad/prioridad

En esta etapa no hay que crear un esquema de sincronización partiendo de cero, sino modificar el ya existente. Las propiedades de vivacidad del esquema actual se analizarán para detectar posibles problemas de vivacidad y, de acuerdo con el resultado de este análisis, se podrá tomar la decisión de modificar el código enriqueciendo el estado del objeto protegido o introduciendo mecanismos de desbloqueo explícito que no rompan las propiedades de seguridad ya conseguidas en la etapa anterior.

4.3.1. Análisis de los desbloqueos

Lo primero es analizar la situación existente. Una herramienta útil para ello, y que servirá también en el posterior diseño, es la *tabla de desbloqueos*. Esta tabla presenta de forma agrupada todas las *situaciones de desbloqueo*, es decir, cuándo tras ejecutarse una operación pasa a cumplirse alguna de las precondiciones de sincronización habilitando el desbloqueo de algún proceso. Puede presentarse en forma matricial, de manera que cada fila corresponda a una operación ejecutada, y cada columna a una situación de bloqueo (CPRE). La tabla 4 muestra la estructura general de estas tablas.

[11] Ada 95 proporciona un mecanismo de replicación de entradas (*entry families*) que sirve precisamente para facilitar el desglose de condiciones de bloqueo. Se puede ver un ejemplo en la sec. 5.2.

Operación ejecutada POST ampliada	**Operación bloqueada CPRE**			
	operA precondición1	operB precondición2	... ...	operX precondiciónN
operación1 POST1	csimple11	csimple12	...	csimple1N
operación12 POST2	csimple21	csimple22	...	csimple2N
... ...	... ...	... ...	csimpleIJ	... ...
operaciónN1 POSTN	csimpleN1	csimpleN2	...	csimpleNN

Cuadro 4: Tabla de sincronización (desbloqueo).

Operación ejecutada POST ampliada	**Operación bloqueada CPRE**	
	Poner $n < MAX$	Tomar $n > 0$
Poner $n^{sal} = n^{ent} + 1(\text{post}) \wedge n^{ent} \geq 0(\text{inv.})$	$n < MAX$ (*)	Cierto
Tomar $n^{sal} = n^{ent} - 1(\text{post}) \wedge n^{ent} \leq MAX(\text{inv.})$	Cierto	$n > 0$ (*)

Cuadro 5: Tabla de desbloqueo para el buffer acotado.

En la cabecera de cada fila se nombra la operación que acaba de ejecutarse y las condiciones que se garantizan a su terminación, es decir,

$$INV \wedge POST_i$$

que denominaremos *POST ampliada*. En la cabecera de cada columna se nombran la operación y la expresión de la precondición de sincronización asociada.

Cada casilla central de la tabla contiene una condición que habría de darse *además de la POST*, que ya está garantizada, para asegurar que se cumple la precondición de sincronización correspondiente a su columna. Esta condición suele denominarse *condición de desbloqueo simplificada* precisamente porque la POST ya está "descontada".

El requisito que se exige a la condición simplificada $csimple_{ij}$ es ser la condición más débil que cumple

$$INV \wedge POST_i \wedge csimple_{ij} \rightarrow CPRE_j$$

Las condiciones de desbloqueo pueden debilitarse aún más teniendo en cuenta no sólo la postcondición individual de cada operación, sino también otras condiciones que estén garantizadas, tales como la invariante del recurso o las que se deriven de las restricciones de uso, historia pasada del recurso, número de procesos en el sistema, etc.

El análisis de la tabla puede servir para revelar que una determinada precondición de sincronización no puede cumplirse en ningún caso inmediatamente después de determinada operación, suponiendo que haya procesos esperando, o, por contra, situaciones no deterministas, en las que diferentes operaciones pueden desbloquearse.

Lo primero (la escasez de situaciones de desbloqueo) podría ser causa de problemas de interbloqueo y lo segundo de situaciones de inanición, prioridades no respetadas, etc.

Operación ejecutada POST (ampliada)	**Condición de desbloqueo (simplificada)**	**CPRE**	**Operación**
operación1 POST1	csimple11	precondición1	operA
	csimple12	precondición2	operB
	...	...	...
	csimple1N1	precondiciónN1	operX
operación2 POST2	csimple11	precondición1	operA
	csimple12	precondición2	operB
	...	...	...
	csimple1N2	precondiciónN2	operX
...	...	...	...
operaciónM POSTM	csimple11	precondición1	operA
	csimple12	precondición2	operB
	...	...	...
	csimple1NM	precondiciónNM	operX

Cuadro 6: Tabla de desbloqueo, disposición alternativa.

En el ejemplo del buffer acotado la tabla de desbloqueos quedaría como muestra la tabla 5. Se puede ver que después de Poner *siempre* puede desbloquearse un proceso que estuviera esperando para Tomar y viceversa. Sin embargo, no siempre se puede Poner tras Poner o Tomar tras Tomar.

No obstante, si tuviéramos la información adicional de que sólo hay un proceso que toma o que pone, dejarían de tener sentido las situaciones de desbloqueo de la diagonal (casillas marcadas con asterisco).

Si el número de condiciones diferentes es relativamente grande, la disposición matricial de la tabla de desbloqueos hará que el ancho sea muy grande. Otra posible disposición es colocar verticalmente una sobre otra las casillas correspondientes a una misma fila. En este caso cada línea de la tabla corresponderá a la combinación de una operación ejecutada con una precondición de sincronización. La tabla podría tener entonces la distribución que se muestra en la tabla 6.

Si las expresiones de las precondiciones de sincronización son complicadas, se podría omitir la correspondiente columna, en la que aparecen repetidas varias veces, asumiendo que esa información ya aparece en la tabla de bloqueos.

4.3.2. Análisis mediante grafos de estados y transiciones

Una alternativa al análisis directo de la tabla de desbloqueos es el uso de un grafo que refleje claramente los estados del recurso compartido y las transiciones entre estos estados. Este grafo corresponde a una máquina de estados (posiblemente infinita) que permite comprender mejor la evolución del recurso ante la invocación de las operaciones definidas sobre él. Los nodos del grafo se corresponden con estados diferentes del recurso y los arcos entre nodos con llamadas a operaciones que provocan el cambio a un estado diferente—es decir, transiciones. Los arcos estarán etiquetados con los nombres de las operaciones a realizar (adjuntando, si es necesario, los parámetros que requiera el realizar esa transición) y los nodos tendrán asociado el estado correspondiente del recurso. De cada nodo saldrán arcos a los nodos que representan estados alcanzables según determinen las condiciones de sincronización y el protocolo de llamadas a respetar.

Más formalmente, por estado entendemos una fórmula lógica definida sobre el dominio del recurso — p.ej. $numCoches = MAX$, $escritores = 0$, etc. Para que un grafo de estados sea de utilidad para modelar la evolución en el tiempo de un recurso, exigiremos que

- en un momento dado, el recurso sólo pueda estar **en uno de los estados del grafo**, y

- si estando el recurso en un estado E del grafo es posible ejecutar una de sus operaciones – es decir, pueden ser ciertas simultáneamente E y la PRE y CPRE de la operación y lo permiten los protocolos de llamada, número de procesos, etc. – existirá un arco etiquetado con la llamada a dicha operación con origen en E y llegada a los estados compatibles con la aplicación de la POST de la operación.

A pesar de estas restricciones, disponemos de bastante libertad a la hora de definir los grafos de estados de un recurso, como se verá en los siguientes ejemplos.

La figura 2 muestra un grafo de estados correspondiente a un *buffer* acotado. El único estado que se representa es el número de elementos, ya que los datos contenidos en el *buffer* no son relevantes para las condiciones de sincronización. Es decir, los estados son las fórmulas

$$n = 0, n = 1, \ldots, n = MAX - 1, n = MAX$$

si bien la representación gráfica de los nodos suele ser más esquemática si ello no lleva a confusión. En este ejemplo también se ha hecho abstracción de los parámetros de las operaciones, al resultar irrelevantes para el análisis.

En la figura se puede ver claramente que la evolución de los estados se debe a la acción de operaciones `Poner` y `Tomar`. Una sucesión suficientemente larga de operaciones `Poner` llevará a un estado final de *buffer* lleno, donde la única operación posible es `Tomar`; lo propio ocurre con una serie de operaciones `Tomar`, que llevará el estado del recurso hacia el extremo opuesto del grafo. Por tanto estaremos en un situación en la que no hay un problema de vivacidad, ya que no es posible que se realice un número arbitrariamente grande de operaciones de un tipo sin dar oportunidad a una operación del otro.

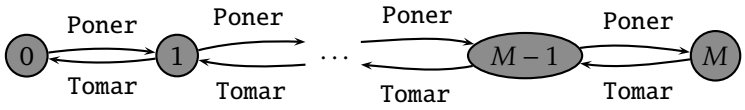

Figura 2: Grafo de estados y transiciones para un buffer

Otro ejemplo de análisis es el del *buffer* par/impar, en el que, habiendo capacidad para un solo elemento, la operación de retirar puede elegir entre un número par o impar, bloqueándose si no hay dato disponible o si no es de la paridad requerida. El grafo se muestra en la figura 3 (ver sección 5.2 y los apuntes de especificación). En un sentido estricto, el número de estados posibles es infinito, por los infinitos valores que puede tomar el dato almacenado y que es necesario para realizar la sincronización. Sin embargo las precondiciones de la operación `Tomar` en este caso sólo distinguen la paridad del dato; por ello podemos interpretar el valor concreto y quedarnos sólo con representantes de las clases de datos que permitirían desbloquear o no una operación en particular. Los estados del grafo son, pues, las fórmulas:

$$\neg HayDato, HayDato \wedge Dato \bmod 2 = 0, HayDato \wedge Dato \bmod 2 = 1$$

Ahora los parámetros de las operaciones sí son relevantes, si bien se han sustituido los valores concretos por patrones de llamada (par, impar) suficientemente representativos.

Al igual que en el caso de *buffer* acotado, no hay problemas de vivacidad.

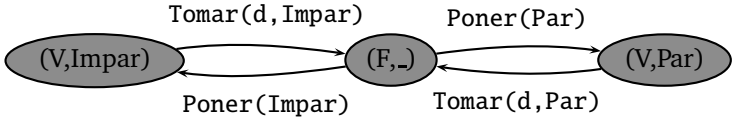

Figura 3: Evolución del *buffer* de par o impar

Un tercer ejemplo es el proporcionado por el problema del gestor de los lectores y escritores, en la figura 4 (ver sección 5.3). En este caso el tamaño del grafo es, de nuevo, potencialmente infinito, y las condiciones de sincronización no ofrecen ninguna pista sobre cómo *abstraer* el estado del recurso para tener una instancia finita y representativa del mismo.

Sin embargo, como suele suceder, el número de procesos en el sistema, aunque desconocido de antemano, es finito, y este número de procesos queda en parte reflejado en el recurso. En nuestro caso, el número de lectores, potencialmente ilimitado, tiene una cota superior en el número verdadero de procesos lectores presentes en el sistema. Fijando este a una cantidad tendremos un grafo finito (o, más bien, una familia infinita de grafos finitos) con el que poder aproximarnos al comportamiento del gestor.

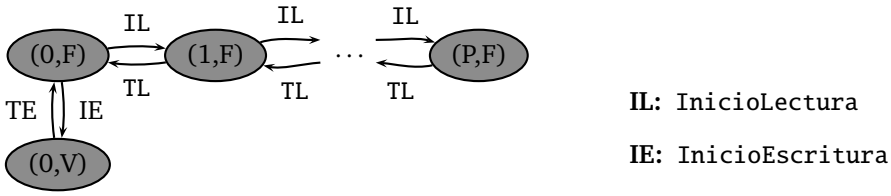

Figura 4: Grafo de lectores y escritores

En este caso existen itinerarios dentro del grafo que recorren un conjunto de nodos (pongamos, el comprendido por aquel etiquetado con (1, F) y el que lo estaría con (2, F)) que, sin violar ninguna propiedad de seguridad, no permiten que se tomen determinadas transiciones (por ejemplo, la que inicia una operación de lectura). En este caso se adivina un problema latente de vivacidad, pues es posible que un sistema real se viese atrapado en un ciclo de este tipo, en que los lectores parecen *aliarse* entre sí para no dejar entrar a ningún escritor. En efecto, esto es lo que suele verse en las ejecuciones con implementaciones reales que se limitan a respetar las condiciones de seguridad.

La posibilidad de este tipo de comportamientos se detecta, en general, por la existencia de ciclos en el grafo que pueden ser recorridos un número ilimitado de veces y que excluyen a alguna (o algunas) de las transiciones—es decir, que no permiten la aplicación de determinadas operaciones al recurso. Cabe señalar que la existencia de este tipo de ciclos al nivel de abstracción proporcionado por el grafo no determina necesariamente que vayan a manifestarse problemas de vivacidad: es común que, en la práctica, muchos problemas de vivacidad latentes no aparezcan, pero ello no impide tenerlos en consideración y, en lo posible, evitarlos. Por otro lado, es necesario realizar un análisis más fino ante la posibilidad de problemas de vivacidad. En determinados casos no hay, en el fondo, tal problema, pues el entrelazado de los procesos, que puede estar parcialmente forzado por el número de ellos en el sistema, pausas obligatorias que excluyen a un proceso de ser planificado, determinadas relaciones de prioridad que no están completamente recogidas en el grafo o consideraciones del protocolo exigido de llamadas, hace que algunos ciclos del grafo no planteen ningún problema.

4.3.3. Refinamiento de la interacción

Tras el análisis, la acción. Si en una determinada fila de la tabla de desbloqueos observamos que diferentes situaciones de desbloqueo pueden darse simultáneamente al terminar de ejecutarse una operación, que existen nodos del grafo de los que parten varios arcos (no excluyentes, ver figura 8) hacia nodos diferentes o ciclos que excluyen a determinadas operaciones, o ciclos en el grafo de estados de la naturaleza que acabamos de mencionar, podemos estar ante un problema potencial de vivacidad.

Una vía para resolver este tipo de problemas, y que suele ser suficiente para muchos casos prácticos (ver 5.4) consiste en *aumentar el estado interno del recurso* de tal manera que se puedan deshacer empates entre las diferentes situaciones de desbloqueo.

Operación	CPRE informal	¿Dep.?	CPRE codificada	Condition
Op (...)	CPRE	sí	$\Phi[a \mapsto x_1]$	q_1
			$\Phi[a \mapsto x_2]$	q_2
			⋮	⋮
			$\Phi[a \mapsto x_n]$	q_n

Cuadro 7: Tabla de bloqueos con identificadores de condición.

4.3.4. Introducción de desbloqueos explícitos

Llevando al extremo la idea de enriquecer el estado del objeto protegido se puede llegar a un esquema general en el que todos los desbloqueos estén gobernados por el valor que en un momento dado toman una serie de variables auxiliares, sin que se produzca ninguna situación no determinista.

Comentaremos aquí la estrategia conducente a diseñar una solución con desbloqueos explícitos *pura*, pero en la práctica pueden darse situaciones en las que lo más sencillo sea optar por una solución *mixta*, en la que algunos desbloqueos son automáticos y sólo algunos se implementan mediante desbloqueos explícitos.

En cualquier caso, hemos de recalcar que la técnica de los desbloqueos explícitos es más complicada que los mecanismos de sincronización automática para los que están pensados los objetos protegidos y, además, si se usan mal, *puede introducir situaciones indeseables* – como interbloqueos – que difícilmente se darían con la sincronización automática.

El motivo de explicar esta técnica es que hay muchos lenguajes de programación que no proporcionan mecanismos de sincronización automática y porque, como veremos en el párrafo 5.4, los desbloqueos explícitos pueden estar justificados por razones de eficiencia.

La técnica se basa, como ya hemos visto en la sección 3.6, en introducir etiquetas que identifican las diferentes situaciones de bloqueo que se desea manejar explícitamente. Para facilitar que el código resultante distinga claramente entre las diferentes fases de desarrollo, se extenderá la tabla de bloqueo con dichas etiquetas, como se muestra en la tabla 7.

Como se explicó en la sección 3.6, el esquema de código consiste en sustituir las CPREs por la condición de que la variable *Siguiente* tome un valor en particular:

```
type Condicion is (Ninguno, Cond1, ..., CondN);
protected type TipoRecurso is
    entradas públicas: dan valor a Siguiente
private
    OpCond1 (parámetros);
    ...
    OpCondN (parámetros);
    ...
    Siguiente : Condicion := Ninguno;
end TipoRecurso;
protected body TipoRecurso is
    ...
    entry OpCond1 (parámetros)
    when Siguiente = Cond1 is
    begin
        Siguiente = Ninguno;
        ...
    end OpCond1;
    ...
    entry OpCondN (parámetros)
```

```
    when Siguiente = CondN is
    begin
        Siguiente = Ninguno;
        ...
    end OpCondN;
end TipoRecurso;
```

El desbloqueo explícito se consigue al forzar un proceso, inmediatamente antes de abandonar el objeto protegido, el valor de *Siguiente*. Ahora el riesgo que se ha de evitar es el de tratar de desbloquear en una entrada *donde no hay procesos bloqueados*. Esto es peligroso, porque podría dejar al objeto protegido en un estado en el que no se cumpla ninguna guarda *a pesar de cumplirse alguna CPRE* y esto podría provocar incluso un interbloqueo.

Para ayudarnos a evitar esta situación indeseable, Ada 95 incluye un atributo que permite consultar, desde el código de alguna entrada, por el número de procesos bloqueados en una determinada guarda:

Operación'Count

devuelve el número de procesos en espera en la cola correspondiente a la guarda de *Operación*.[12]

Con esto, el esquema de código para los desbloqueos desde una operación saliente queda de la siguiente forma:

```
entry OpX (...)
when ... is
    ...
   -- Aquí se cumple la POST
   -- y, por tanto, alguna de las CPREs correspondientes
    if CPRE1 and criterio1 and Operación1'Count >0 then
        Siguiente := Q1;
    elsif CPRE2 and criterio2 and Operación2'Count >0 then
        Siguiente := Q2;
    ...
    elsif CPREN and criterioN and OperaciónN'Count >0 then
        Siguiente := QN;
    end if;
end OpX;
```

5. Ejemplos

5.1. Problema del productor y el consumidor (sincronización automática, independencia de los argumentos de entrada)

Se trata de un problema clásico en que dos procesos cooperan de modo que uno de ellos se encarga de generar uno a uno una serie de datos, y el otro proceso se encarga de procesar dichos datos, también uno a uno, y en el mismo orden.

La idea es que ambos procesos operen concurrentemente. Si el tiempo de generar o procesar un dato es variable, para evitar esperas innecesarias se utiliza un buffer intermedio en que se van almacenando los datos generados por el primer proceso, en espera de ser procesados por el segundo. Se supone que este buffer tiene una capacidad limitada y constante.

[12] Esta posibilidad de Ada 95 no es necesaria, estrictamente hablando: siempre es posible desdoblar la operación en la que se consulta el número de procesos esperando en dos. La primera se limitaría a incrementar, utilizando una variable del estado del objeto, el número de procesos que desean realizar una operación, y la segunda iniciaría de forma efectiva esa operación previa consulta del número de procesos anotado. La aceptación de la operación debe, por supuesto, decrementar el número de procesos en espera.

El buffer constituye un elemento o recurso compartido por los dos procesos. La actualización correcta del buffer al almacenar o extraer elementos exige, en general, que estas operaciones se hagan con acceso exclusivo a los datos internos del buffer, por lo que resulta apropiado programarlo como una abstracción de datos encapsulada como un objeto protegido.

Propiedades de la abstracción de datos

- Los elementos se almacenan uno a uno, y se extraen posteriormente en el mismo orden.

Propiedades de seguridad

- No se puede almacenar y extraer a la vez.
- El productor no puede almacenar en el buffer si éste está lleno.
- El consumidor no puede extraer del buffer si éste está vacío.

En este problema no hay propiedades particulares de vivacidad/prioridad.

Especificación abstracta

Las propiedades anteriores se recogen en la siguiente especificación:

C-TADSOL TipoBufferAcotado
USA TipoCola, TipoElemento
OPERACIONES
ACCIÓN Poner: *TipoBufferAcotado[es]* × *TipoElemento[e]*
ACCIÓN Tomar: *TipoBufferAcotado[es]* × *TipoElemento[s]*

SEMÁNTICA
DOMINIO:
TIPO: *TipoBufferAcotado = (cuantos:*$\mathbb{N}$ × *cola:TipoCola(TipoElemento))*
INVARIANTE: ∀ buf ∈ TipoBufferAcotado • buf.cuantos = NumElementos (buf.cola)
INICIAL(buf**):** buf.cuantos^{sal} = 0 ∧ buf.cola^{sal} = CrearVacia

PRE: Cierto
***CPRE*:** *El buffer no está totalmente lleno.*
CPRE: buf.cuantos < MaxElementos
Poner (buf, elemento)
***POST*:** *El buffer tiene un elemento mas, situado al final de la cola*
POST: buf.cuantos^{sal} = buf.cuantos^{ent} + 1∧
buf.cola^{sal} = Insertar(buf.cola^{ent}, elemento)

PRE: Cierto
***CPRE*:** *El buffer no está vacío.*
CPRE: buf.cuantos > 0
Tomar (buf, elemento)
***POST*:** *El buffer tiene un elemento menos, retirado del principio de la cola*
POST: buf.cuantos^{sal} = buf.cuantos^{ent} - 1∧ elemento^{sal} = Primero(buf.cola^{ent}) ∧
buf.cola^{sal} = Borrar(buf.cola^{ent})

Esta especificación corresponde directamente a una implementación con objetos protegidos.

Estructura del programa

El programa constará de dos procesos, Productor y Consumidor, y un objeto protegido, BufferAcotado, accesible desde ambos. El primer proceso entrega sucesivamente elementos al buffer y el segundo los va tomando.

Los procesos podrán adoptar la forma:

```
BufferAcotado : TipoBufferAcotado;
...
task type TProductor;
task body TProductor is
   Dato : TipoElemento;
begin
   loop
      <producir dato>
      BufferAcotado.Poner (Dato);
   end loop;
end TProductor;

task type TConsumidor;
task body TConsumidor is
   Dato : TipoElemento;
begin
   loop
      BufferAcotado.Tomar (Dato);
      <procesar dato>
   end loop
end TConsumidor;

Productor : TProductor;
Consumidor : TConsumidor;
```

Definición de la abstracción de datos

La definición de la abstracción de datos viene dada por las necesidades de comunicación de los procesos, que como hemos visto se reduce a las dos operaciones Poner y Tomar.

```
with TipoElemento;
use TipoElemento;

protected type TipoBufferAcotado is
  entry Poner (Item : in TipoElemento);
  entry Tomar (Item: out TipoElemento);
private
  ...
end TipoBufferAcotado;
```

Implementación de la abstracción de datos

Del enunciado resulta inmediato que el comportamiento del buffer es igual que el de una cola FIFO. Si se realiza la abstracción de datos en términos de un vector usado en forma circular, el refinamiento incial del recurso resulta:

```
protected type TipoBuferAcotado is
  entry Poner (Item : in TipoElemento);
```

```
  entry Tomar (Item: out TipoElemento);
private
  MAX : constant Positive := ...;
  Sale : Natural := 0;
  Entra : Natural := 0;
  Datos : array (0..MAX-1) of TipoElemento;
end TipoBufferAcotado;
protected body TipoBuferAcotado is
  entry Poner (Item : in TipoElemento) when ... is
  begin
    Datos (Entra) := Item;
    Entra := (Entra + 1) mod MAX;
  end Poner;
  entry Tomar (Item: out TipoElemento) when ... is
  begin
    Item := Datos (Sale);
    Sale := (Sale + 1) mod MAX;
  end Tomar;
end TipoBufferAcotado;
```

Diseño de la sincronización (de seguridad)

El enunciado distinguía tres restricciones de seguridad. La primera de ellas, la exclusión entre operaciones, es implícita en el objeto protegido. Las otras dos se recogen en la tabla de sincronización correspondiente a las situaciones de bloqueo:

Operación	CPRE informal	dep.	CPRE codificada
Poner (Item : TDato[ent])	Buffer no lleno	no	$n < MAX$
Tomar (Item : TDato[sal])	Buffer no vacío	no	$n > 0$

Para codificar las condiciones se ha introducido la variable de estado, n, que habrá de ser actualizada en las operaciones del tipo protegido.

Implementación de la sincronización (de seguridad)

La integración de la sincronización de seguridad en el esquema inicial de código es inmediata, al no haber ninguna dependencia de datos de entrada.

```
protected type TipoBuferAcotado is
  entry Poner (Item : in TipoElemento);
  entry Tomar (Item: out TipoElemento);
private
  MAX : constant Positive := ...;
  Sale : Positive := 0;
  Entra : Positive := 0;
  Datos : array (0..MAX-1) of TipoElemento;
  N : Positive := 0; -- no. de elementos
end TipoBufferAcotado;
protected body TipoBuferAcotado is
  entry Poner (Item : in TipoElemento)
  when N < MAX is
  begin
    Datos (Entra) := Item;
    Entra := (Entra + 1) mod MAX;
```

Operación ejecutada POST ampliada	**Operación bloqueada CPRE**	
	Poner	Tomar
	$n < MAX$	$n > 0$
Poner $n^{sal} = n^{ent} + 1$(post) $\wedge\ n^{ent} \geq 0$(inv.)	$n < MAX$	Cierto
Tomar $n^{sal} = n^{ent} - 1$(post) $\wedge\ n^{ent} \leq MAX$(inv.)	Cierto	$n > 0$

Cuadro 8: Tabla de desbloqueo para el buffer acotado.

```
    N := N+1;
  end Poner;
  entry Tomar (Item: out TipoElemento)
  when N > 0 is
  begin
    Item := Datos (Sale);
    Sale := (Sale + 1) mod MAX;
    N := N-1;
  end Tomar;
end TipoBufferAcotado;
```

El análisis de la tabla de desbloqueos (tabla 8) y la ausencia de requisitos de vivacidad adicionales hace que no sea necesario retocar este código.

Obsérvese, en efecto, que si por algún motivo hubiera más probabilidades de atender las llamadas de *Poner* que las de *Tomar*, la precondición de aquellas se acabaría haciendo falsa, y viceversa.

5.2. Buffer de pares e impares (sincronización automática, bloqueo en dos fases)

Este problema es una sencilla variación de la idea del productor/consumidor en la que los consumidores esperan por datos que cumplan una determinada propiedad. En su versión más simple, el buffer tiene capacidad para un único dato numérico y los productores son de dos tipos: los que esperan por un dato de valor par y los que esperan por un dato impar.

Especificación abstracta del recurso

C-TADSOL BufferPI

OPERACIONES
ACCIÓN Poner: *Tipo_Buffer_PI[es]* $\times$ *Tipo_Dato[e]*
ACCIÓN Tomar: *Tipo_Buffer_PI[es]* $\times$ *Tipo_Dato[s]* $\times$ *Tipo_Paridad[e]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Buffer_PI = (dato : Tipo_Dato* $\times$ *hayDato : Booleano)*
Tipo_Paridad = par|impar
Tipo_Dato $= \mathbb{N}$
INICIAL(b**):** ¬b.hayDato

CPRE*: *El buffer no está lleno
CPRE: ¬b.hayDato

Poner (b, d)
***POST*:** *Añadimos un elemento al buffer*
POST: b^{sal}.hayDato $\wedge$ b^{sal}.dato = d

***CPRE*:** *El buffer no está vacío y el dato es del tipo que requerimos*
CPRE: b.hayDato $\wedge$ Concuerda(b.dato, t)
DONDE: Concuerda (d, t) $\equiv$ (d *mod* 2 = 0 $\leftrightarrow$ t = par)
Tomar (b, d, t)
***POST*:** *Retiramos el elemento del buffer*
POST: $\neg b^{sal}$.hayDato $\wedge$ d = b^{ent}.dato

Obsérvese cómo, a diferencia del ejemplo anterior, la CPRE de la operación *Tomar* depende de uno de los datos de entrada.

Estructura del programa

Se tendrán procesos de dos tipos: *Productor* y *Consumidor*. A su vez, los consumidores se subtipan de acuerdo con que requieran datos pares o impares.

Existirá un objeto protegido *Buffer* para realizar la interacción entre productores y consumidores.

La forma de los procesos será, pues:

```
type TPeticion is (Par, Impar);
...
Buffer : TBuffer;
...
task type TProductor;
task body TProductor is
begin
  loop
    < producir Dato >
    Buffer.Poner (Dato);
  end loop;
end TProductor;

task type TConsumidor(Que_Tipo : TPeticion);
task body TConsumidor is
  Dato: TDato;
begin
  loop
    Buffer.Tomar (Que_Tipo, Dato);
    < consumir Dato >
  end loop;
end TConsumidor;

-- Los productores y consumidores se inician sólo por declararlos
-- como componentes de un vector.
Consumidores_Par : array(1..Num_Consumidores/2) of TConsumidor(Par);
Consumidores_Impar : array(1..Num_Consumidores/2) of TConsumidor(Impar);
Productores : array(1..Num_Productores) of TProductor;
```

Definición de la abstracción de datos

Acabamos de ver que las necesidades de productores y consumidores se reducen a las dos operaciones *Poner* y *Tomar*:

```
protected type TBuffer is
  entry Tomar (Pet : in TPeticion; Que : out TDato);
  entry Poner (Que : in TDato);
private
  Dato : TDato;
  ...
end TBuffer;
```

Diseño de la sincronización de seguridad

La tabla de bloqueos de este problema se detalla a continuación. Por simplicidad omitimos el parámetro correspondiente al estado del propio objeto protegido:

Operación	**CPRE informal**	**dep.**	**CPRE codificada**
Poner (Que : TDato[ent])	Buffer no lleno	no	$\neg HayDato$
Tomar (Pet : TPeticion[ent]; Que : TDato[sal])	Hay dato y es de tipo *Pet*	**sí**	$HayDato \wedge (Dato \mod 2 = 0)$
			$HayDato \wedge (Dato \mod 2 = 1)$

Obsérvese cómo se ha realizado el desglose de la CPRE de la operación de *Tomar* dependiendo de que *Pet* tome el valor *Par* o *Impar*. De no hacerse así el diseño sería incorrecto, pues nos llevaría a una solución con riesgo de interbloqueo: el buffer con un dato par, y un consumidor impar bloqueando a los consumidores pares.

Implementación de la sincronización de seguridad

Como acabamos de ver, es necesario desdoblar la operación *Tomar* pues los bloqueos de los consumidores de uno y otro tipo deben ser independientes.

Para ello aplicamos el esquema de código para operaciones con CPRE dependiente de algún parámetro de entrada, en el que la guarda de la entrada pública está siempre abierta y se efectúa un *requeue* en una de las entradas privadas.

```
protected type TBuffer is
  entry Tomar (Pet : in TPeticion; Que : out TDato);
  entry Poner (Que : in TDato);
private
  Dato : TDato;
  Hay_Dato : Boolean := False;
  entry Tomar_Aplazado_Par (Pet : in TPeticion; Que : out TDato);
  entry Tomar_Aplazado_Impar (Pet : in TPeticion; Que : out TDato);
end TBuffer;

protected body TBuffer is

  entry Tomar (Pet : in TPeticion; Que : out TDato)
  when True is
  begin
    if Pet = Par then
      requeue Tomar_Aplazado_Par;
    else
      requeue Tomar_Aplazado_Impar;
```

```
    end if;
  end Tomar;

  entry Tomar_Aplazado_Par (Pet : in TPeticion; Que : out TDato)
  when Hay_Dato and Dato mod 2 = 0 is
  begin
    Que := Dato;
    Hay_Dato := False;
  end Tomar_Aplazado_Par;

  entry Tomar_Aplazado_Impar (Pet : in TPeticion; Que : out TDato)
  when Hay_Dato and Dato mod 2 = 1 is
  begin
    Que:= Dato;
    Hay_Dato:= False;
  end Tomar_Aplazado_Impar;

  entry Poner (Que : in TDato)
  when not Hay_Dato is
  begin
    Dato := Que;
    Hay_Dato := True;
  end Poner;
end TBuffer;
```

5.2.1. Versión con familia de entradas

Como se comentó anteriormente, Ada 95 proporciona un mecanismo de replicación de entradas – *entry families* – que facilita la implementación del desglose de operaciones.

Ahora tendríamos, en lugar de dos operaciones privadas *Tomar_Aplazado_Pares* y *Tomar_Aplazado_Impares* declaradas explícitamente, una familia de operaciones privadas *Tomar_Aplazado* indexada por el tipo *TPeticion*.

El código del objeto protegido quedaría de la siguiente manera:

```
protected type TBuffer is
  entry Tomar (Pet : in TPeticion; Que : out TDato);
  entry Poner (Que : in TDato);
private
  Dato : TDato;
  Hay_Dato : Boolean := False;
  entry Tomar_Aplazado
    (Tpeticion)
    (Pet : in TPeticion; Que : out TDato);
end TBuffer;

protected body TBuffer is

  entry Tomar (Pet : in TPeticion; Que : out TDato)
  when True is
  begin
    requeue Tomar_Aplazado (Pet);
  end Tomar;

  entry Tomar_Aplazado
```

Operación ejecutada / **POST ampliada**	**Operación bloqueada** **CPRE**		
	Poner ¬HayDato	Tomar_Aplazado(Par) HayDato ∧ Dato mod 2 = 0	Tomar_Aplazado(Impar) HayDato ∧ Dato mod 2 = 1
Poner (Que) HayDato ∧ Dato = Que	Falso	Dato mod 2 = 0	Dato mod 2 = 1
Tomar (Par) (Pet, Que) ¬HayDato ∧ Que = Dato^{ent}	Cierto	Falso	Falso
Tomar (Impar) (Pet, Que) ¬HayDato ∧ Que = Dato^{ent}	Cierto	Falso	Falso

Cuadro 9: Tabla de desbloqueos para el buffer de pares/impares con replicación de entradas.

```
    (for P in TPeticion)
    (Pet : in TPeticion; Que : out TDato)
  when Hay_Dato and ((Dato mod 2 = 0) = (P = Par)) is
  begin
    Que := Dato;
    Hay_Dato := False;
  end Tomar_Aplazado;

  entry Poner (Que : in TDato)
  when not Hay_Dato is
  begin
    Dato := Que;
    Hay_Dato := True;
  end Poner;
end TBuffer;
```

La tabla de desbloqueos para esta solución no evidencia ninguna situación indeseable. La tabla se muestra en la tabla 9.

Se puede observar que se ha aplicado la simplificación de no mostrar la operación pública *Tomar*, cosa que se hará, en general, cuando una operación del recurso dependa de algún parámetro de entrada. Esto es debido a que, de acuerdo con el esquema de código que se aplica en estos casos, la entrada pública *no puede efectuar ninguna modificación en el estado del objeto protegido que ocasione un desbloqueo*.

5.3. Problema de los lectores y los escritores (gestor de sincronización)

El problema de los lectores y los escritores es un paradigma clásico en Programación Concurrente. En dicho problema se plantea la necesidad de gestionar el acceso simultáneo a un determinado recurso (en este caso cierta información, similar a una base de datos) por parte de diferentes tipos de procesos. Los procesos realizan dos clases de operaciones sobre la información del recurso: las operaciones de *lectura* no modifican para nada dicha información, mientras que las operaciones de *escritura* sí pueden modificarla.

Las condiciones generales de acceso simultáneo son que varias operaciones de lectura pueden hacerse a la vez, pero que una operación de escritura debe hacerse de forma exclusiva respecto a cualquier otra operación, ya sea lectura o escritura.

Especificación abstracta del recurso

Puesto que hay casos en que varios procesos acceden simultáneamente al recurso, el objeto protegido no puede encapsular dicho recurso como abstracción de datos, sino limitarse a conceder

los permisos de acceso adecuados. El código de manipulación del recurso será externo al objeto protegido, siendo éste un mero gestor de sincronización.

C-TADSOL TipoGestor

OPERACIONES
ACCIÓN IniciarLectura: *TipoGestor[es]*
ACCIÓN TerminarLectura: *TipoGestor[es]*
ACCIÓN IniciarEscritura: *TipoGestor[es]*
ACCIÓN TerminarEscritura: *TipoGestor[es]*

PROTOCOLOS: Leer: IniciarLectura ; TerminarLectura
Escribir: IniciarEscritura ; TerminarEscritura

CONCURRENCIA: Leer* – concurrencia aparente

SEMÁNTICA
DOMINIO:
TIPO: *TipoGestor = (numLectores :* $\mathbb{N} \times$ *escribiendo : Booleano)*
INVARIANTE: $\forall$ g $\in$ TipoGestor • $\neg$g.escribiendo $\vee$ g.numLectores = 0
INICIAL(g): $\neg$g.escribiendo $\wedge$ g.numLectores = 0

CPRE: $\neg$g.escribiendo
IniciarLectura (g)
POST: $g^{sal} = g^{ent} \setminus g^{sal}$.numLectores = g^{ent}.numLectores + 1

CPRE: Cierto
TerminarLectura (g)
POST: $g^{sal} = g^{ent} \setminus g^{sal}$.numLectores = g^{ent}.numLectores - 1

CPRE: g.numLectores = 0 $\wedge\neg$ g.escribiendo
IniciarEscritura (g)
POST: g^{sal}.escribiendo

CPRE: Cierto
TerminarEscritura (g)
POST: $\neg g^{sal}$.escribiendo

La concurrencia de transacciones de lectura se consigue simplemente permitiendo el entrelazado de las operaciones de inicio y terminación. No implica concurrencia real entre operaciones sobre el gestor. La especificación del gestor puede traducirse entonces a una implementación con objetos protegidos.

Propiedades de la abstracción de datos

Como se ve en la especificación, el objeto protegido no gestiona directamente el recurso, sino los permisos de acceso. Las operaciones sobre los permisos serán las necesarias para iniciar y terminar una lectura o una escritura. El objeto debe mantener actualizada la información apropiada para decidir cuándo se puede conceder un permiso determinado para iniciar una cierta operación. Dicha información será, básicamente, el número de procesos que están usando el recurso en cada modo de operación.

Propiedades de seguridad

- Un escritor no puede acceder al recurso si hay algún otro proceso leyendo o escribiendo.
- Un lector no puede acceder al recurso si hay algún proceso escribiendo.

Estructura del programa

El programa constará de dos tipos de procesos, *TLector* y *TEscritor*, y un objeto protegido *Permiso* accesible por ambos. El código de los lectores y escritores seguirá un esquema como el siguiente:

```
Permiso : TipoGestor;
...
task type TLector;
task body TLector is
begin
  loop
    ...
    Permiso.IniciarLectura;
    <leer datos>
    Permiso.TerminarLectura;
    ...
  end loop;
end TLector;

task type TEscritor;
task body TEscritor is
body
  loop
    ...
    Permiso.IniciarEscritura;
    <modificar datos>
    Permiso.TerminarEscritura;
    ...
  end loop;
end TEscritor;
```

Si las operaciones «*leer datos*» y «*modificar datos*» fuesen acciones simples, se podrían encapsular junto con las correspondientes de Iniciar y Terminar en una sola operación (Leer y Escribir) sobre la base de datos. En caso contrario las tareas clientes deberán responsabilizarse de seguir el protocolo de uso que se indica.

Definición de la abstracción de datos

Se definen las cuatro operaciones: IniciarLectura, TerminarLectura, IniciarEscritura y TerminarEscritura.

```
protected type TipoGestor is
  entry IniciarLectura;
  entry IniciarEscritura;
  entry TerminarLectura;
  entry TerminarEscritura;
private
  ...
end TipoGestor;
```

Implementación de la abstracción de datos

El estado del gestor debe llevar la contabilidad de los permisos de acceso. Dado que varios lectores pueden conseguir el permiso de acceso a la vez, y en cambio un escritor debe hacerlo completamente en solitario, la contabilidad de los permisos de los lectores y de los escritores es diferente. En el caso de los escritores basta con una variable booleana que indique si hay o no el único escritor posible; para el caso de los lectores es necesaria una variable de tipo Natural que controle cuántos lectores tienen permiso en cada momento.

```
protected type TipoGestor is
  entry IniciarLectura;
  entry IniciarEscritura;
  entry TerminarLectura;
  entry TerminarEscritura;
private
  Escribiendo : Boolean := False;
  NumLectores : Natural := 0;
end TipoGestor;
protected body TipoGestor is
  entry IniciarLectura
  when ... is
  begin
    NumLectores := NumLectores + 1;
  end IniciarLectura;

  entry IniciarEscritura
  when ... is
  begin
    Escribiendo := True;
  end IniciarEscritura;

  entry TerminarLectura
  when ... is
  begin
    NumLectores := NumLectores - 1;
  end TerminarLectura;

  entry TerminarEscritura
  when True is
  begin
    Escribiendo := False;
  end;
end TipoGestor;
```

Diseño de la sincronización (de seguridad)

El enunciado distinguía dos restricciones de seguridad que se corresponden con las siguientes situaciones de espera:

Operación	CPRE (informal)	¿dep.?	CPRE codificada
IniciarLectura	ningún escritor	no	not Escribiendo
IniciarEscritura	ningún escritor y ningún lector	no	not Escribiendo and (NumLectores = 0)
TerminarLectura	—	—	—
TerminarEscritura	—	—	—

Las operaciones que pueden hacer ciertas las precondiciones de sincronización se recogen en la siguiente tabla:

Operación ejecutada **POST ampliada**	**Operación CPRE**	
	IniciarLectura ¬escribiendo	IniciarEscritura ¬escribiendo ∧ (numLectores=0)
IniciarLectura ¬ escribiendo ∧ numLectores^{sal} = numLectores^{ent} + 1	Cierto	Falso
IniciarEscritura numLectores = 0 ∧ escribiendo	Falso	Falso
TerminarLectura numLectores^{sal} = numLectores^{ent} - 1 ∧ ¬escribiendo	Cierto	numLectores = 0
TerminarEscritura ¬escribiendo ∧ numLectores = 0	Cierto	Cierto

Analizando las condiciones de desbloqueo de la tabla anterior, debemos hacer las siguientes consideraciones:

1. La reactivación de un proceso Escritor que estuviera esperando para escribir puede darse, bien porque termina el último lector (si estabamos en una fase de lectura), bien porque termina el escritor (caso de fase de escritura).

2. La reactivación de un proceso Lector que estuviera esperando para leer puede darse cuando termine el escritor que provocó la parada. Obsérvese que la tabla permite que entren consecutivamente todos los lectores que estuvieran esperando, ya que al completarse la operación de IniciarEscritura se sigue haciendo cierta la guarda de dicha operación.

Implementación de la sincronización (de seguridad)

Las condiciones de bloqueo y desbloqueo recogidas en las tablas anteriores pueden incorporarse al código del objeto protegido. Así se tendría el refinamiento siguiente:

```
protected type TipoGestor is
  entry IniciarLectura;
  entry IniciarEscritura;
  entry TerminarLectura;
  entry TerminarEscritura;
private
  Escribiendo : Boolean := false;
  NumLectores : Positive := 0;
end TipoGestor;
protected body TipoGestor is
  entry IniciarLectura
  when not Escribiendo is
  begin
    NumLectores := NumLectores + 1;
  end IniciarLectura;

  entry IniciarEscritura
  when not Escribiendo and NumLectores = 0 is
  begin
    Escribiendo := true;
  end IniciarEscritura;
```

```
  entry TerminarLectura is
  begin
    NumLectores := NumLectores - 1;
  end TerminarLectura;

  entry TerminarEscritura is
  begin
    Escribiendo := false;
  end;
end TipoGestor;
```

La implementación del desbloqueo al final de una escritura presenta un cierto problema, ya que se pueden desbloquear tanto lectores como escritores, lo cual implica violar las restricciones de prioridad.

Diseño de la sincronización (de vivacidad/prioridad)

En la tabla de desbloqueos se observa que al final de una operación TerminarEscritura se dan simultáneamente las condiciones para que un Escritor o un Lector puedan ya reanudar su ejecución. Si hay procesos de ambos tipos esperando, ¿cuál de ellos debe continuar? No puede darse continuación a los dos, porque no se respetarían las condiciones de seguridad. Hay que optar por uno u otro, pero, y esto es lo importante, ¿cuál debe ser el criterio?, o lo que es lo mismo, ¿cuál debe ser la prioridad que debamos dar a un proceso frente a otro?.

Lo que se está planteando ahora no son cuestiones que afecten a la consistencia de las posibles soluciones (condiciones de seguridad), sino a todas aquellas cuestiones relativas a la calidad de la ejecución de los procesos del programa. En este sentido, las soluciones deben ser pensadas siguiendo las siguientes consideraciones:

- Que la prioridad puede proponerse o no como una condición inicial del problema que estudiamos.
- Que, salvo que se indique expresamente, se deben evitar aquellas soluciones que presenten problemas generales de vivacidad en su ejecución (inanición, cierre, ...).

El análisis de las condiciones de prioridad suele estar asociado al de las propiedades de vivacidad. En este caso se da una fuerte interdependencia entre ambos aspectos. En efecto, si se decide dar siempre prioridad a los lectores frente a los escritores, puede producirse inanición de éstos últimos si una serie de lectores usan el recurso repetidamente solapando el acceso de unos con el de otros. En el caso inverso, dar prioridad siempre a los escritores frente a los lectores da lugar a una posible inanición de los lectores si hay escritores siempre esperando para usar el recurso.

Por otra parte, la inanición de los escritores puede darse con independencia de la política de prioridades si no se restringe el acceso de nuevos lectores cuando ya hay otros lectores usando el recurso, ya que es posible la situación anteriormente indicada.

Una posible solución es establecer turnos alternativos de prioridad a los lectores y a los escritores. El turno se cambia al terminar una operación de lectura o de escritura, cediendo la preferencia al contrario. Durante el turno de lectura pueden acceder al recurso todos los lectores que estuvieran esperando. Durante el turno de escritura accederá al recurso exactamente un escritor.

Implementación de la sincronización (vivacidad/prioridad)

El establecimiento de turnos alternativos entre lectores y escritores para resolver los problemas de inanición puede verse como un refinamiento en el que se amplía el estado del objeto compartido respecto a su formulación inicial. Una vez introducidas las nuevas variables y modificadas las postcondiciones de las operaciones, se puede comprobar que el riesgo de inanición ha desaparecido.

Operación	CPRE (informal)	¿dep.?	CPRE codificada
IniciarLectura	no escribiendo y, o es turno de leer o no hay escritores esperando	no	¬escribiendo ∧ (IniciarEscritura'Count=0 ∨ turnoLectura)
IniciarEscritura	no escribiendo ni leyendo y, o es turno de escribir, o no hay lectores esperando	no	¬escribiendo ∧ numLectores = 0 ∧ (¬turnoLectura ∨ IniciarLectura'Count = 0)

Operación ejecutada POST ampliada	**Operación CPRE**	
	IniciarLectura ¬escribiendo ∧ (turnoLectura ∨ IniciarEscritura'Count = 0)	IniciarEscritura ¬escribiendo ∧ numLectores = 0 ∧ (¬ turnoLectura ∨ IniciarLectura'Count = 0)
IniciarLectura $\neg escribiendo^{sal} \wedge$ $numLectores^{sal} = numLectores^{ent} + 1$	(turnoLectura ∨ IniciarEscritura'Count = 0)	Falso
IniciarEscritura $numLectores^{sal} = 0 \wedge escribiendo^{sal}$	Falso	Falso
TerminarLectura $numLectores^{sal} = numLectores^{ent} - 1 \wedge$ $\neg escribiendo^{sal} \wedge \neg turnoLectura^{sal}$	IniciarLectura'Count = 0	numLectores = 0
TerminarEscritura $\neg escribiendo^{sal} \wedge numLectores^{sal} = 0$ $turnoLectura^{sal}$	Cierto	IniciarLectura'Count = 0

Cuadro 10: Tablas de bloqueos y desbloqueos para el problema de lectores/escritores con turnos alternativos

Las nuevas tablas de bloqueos y desbloqueos se muestran en la tabla 10.
La codificación del tipo protegido se muestra a continuación:

```
protected type Tipo_Gestor is
   entry IniciarLectura;
   entry IniciarEscritura;
   procedure TerminarLectura;
   procedure TerminarEscritura;
private
   NumLectores : Natural := 0;
   Escribiendo : Boolean := False;
   TurnoLectura : Boolean := False;
end Tipo_Gestor;


protected body Tipo_Gestor is

   entry IniciarLectura when
      not Escribiendo and
      (IniciarEscritura'Count = 0 or TurnoLectura) is
   begin
      Numlectores := Numlectores + 1;
   end IniciarLectura;
```

```
   entry IniciarEscritura when
        Numlectores = 0 and not escribiendo and
        (IniciarLectura'Count = 0 or not TurnoLectura) is
   begin
      Escribiendo := true;
   end IniciarEscritura;

   procedure TerminarLectura is
   begin
      Numlectores := Numlectores - 1;
      TurnoLectura := False;
   end TerminarLectura;

   procedure Terminarescritura is
   begin
      Escribiendo := False;
      TurnoLectura := True;
   end TerminarEscritura;
end Tipo_Gestor;
```

El análisis de esta implementación muestra que en el turno de lectura pueden entrar todos los lectores que estuvieran esperando, o incluso alguno nuevo, pero sólo hasta que termine el primer lector, que cede el turno a los escritores. Por supuesto, los lectores pueden entrar, aunque no sea su turno preferente, si no hay escritores usando el recurso.

Vivacidad/prioridad y grafos de estados

Es ilustrativo observar el aspecto que tienen los grafos de estados en el caso de una implementación, como la anterior, en la que se ha aumentado el estado con consideraciones de vivacidad. Idealmente este grafo nos debería permitir concluir que no hay problemas de vivacidad; como veremos, y de modo comprensible, el aumento del estado del objeto protegido conlleva un aumento del número de estados del recurso y la complejidad del grafo.

Las figuras 5 y 6 muestran, respectivamente, los grafos para el caso de un solo proceso escritor y un solo proceso lector. Los estados están aumentados con el número de lectores o escritores esperando, que causan transiciones de estado no asociadas a operaciones del recurso (tabla 11).[13] Estos dos casos ilustrativos sencillos no deben exhibir problemas de vivacidad, al existir únicamente un proceso de cada tipo.

numLectores: $\mathbb{N}\times$ *Escribiendo:* $\mathbb{B}\times$ *TurnoLect:* $\mathbb{B}\times$ *NumLectEsperando:* $\mathbb{N}\times$ *NumEscritEsperando:* $\mathbb{N}$

Cuadro 11: Descripción de estado

La figura 7 muestra el grafo de estados y transiciones para un lector y un escritor. Es de resaltar la complejidad y aparente falta de estructuración, que hace difícil razonar con él de una manera intuitiva, como se hizo con el de la figura 4. En efecto, en este problema el número de estados del grafo y su complejidad crece mucho con el número de procesos: por ejemplo, con 5 lectores y 5 escritores el grafo ya tiene 312 estados distintos y una estructura casi no discernible. Eso disminuye la aplicabilidad real (y, sobre todo, de forma manual) de los grafos de estados a determinados casos sencillos o, al menos, regulares. No obstante, sí es cierto que en estos casos son una gran ayuda para percibir de qué modo puede evolucionar el estado del recurso.

[13] Esas estarían realmente asociadas a operaciones si se desdoblasen los inicios de lectura/escritura en una petición y un verdadero inicio; la petición únicamente incrementaría el número de procesos en espera.

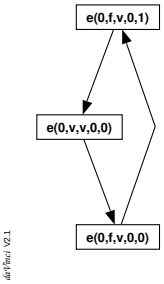

Figura 5: Un solo escritor

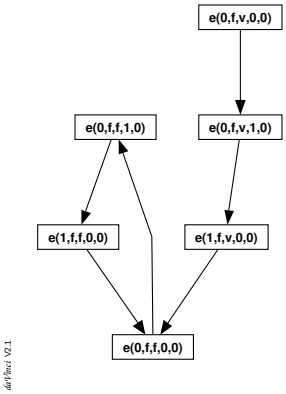

Figura 6: Un solo lector

Como última nota señalaremos que incluso en este caso en que existe información de vivacidad añadida, es posible trabajar con una extensión, un tanto *ad-hoc*, de una máquina de estados. La idea central es restringir el no determinismo en la elección de transiciones (que puede llevar a ciclos indeseados) por un indeterminismo *condicional* que captura la idea de las restricciones de vivacidad impuestas a las condiciones de seguridad. Las transiciones IL anotadas con [1] en la figura 8 deben entenderse como: *si hay algún proceso candidato a tomar una transición IE, entonces no puede tomar la transición IL* (y lo propio con la transición IE anotada con [2]).

Aunque la estructura de este grafo es mucho más clara, también es cierto que el sistema de turnos, traducido por una preferencia en las transiciones, puede, en un caso más complicado, resultar complicado de plasmar. Hay que observar, asimismo, que en el razonamiento sobre este grafo se ha introducido la idea de *"proceso que está esperando a realizar una operación"*, que no había sido necesaria antes. En conclusión, las tablas de desbloqueos ofrecen en general una visión que es más potente, pues permiten recoger de forma explícita todas las condiciones de rearranque, aunque sea más difícil trabajar con ellas. Una conjunción de ambas herramientas permite obtener un análisis y comprensión más completo de muchos sistemas.

5.4. Cuenta bancaria compartida (*sin* y *con* desbloqueos explícitos)

El siguiente ejemplo muestra tres fases en el refinamiento del código y compara ventajas y desventajas de la introducción de desbloqueos explícitos.

Se trata de sincronizar accesos concurrentes a una cuenta bancaria, con las condiciones de seguridad de la exclusión mutua en la actualización del saldo y que éste nunca debe hacerse negativo.

Especificación formal de la interacción

C-TADSOL Cuenta

OPERACIONES
ACCIÓN Ingreso: *Tipo_Cuenta[es]* × *Tipo_Saldo[e]*
ACCIÓN Reintegro: *Tipo_Cuenta[es]* × *Tipo_Saldo[e]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Cuenta = (Saldo : Tipo_Saldo)*
INICIAL(b)**:** b.Saldo = ...

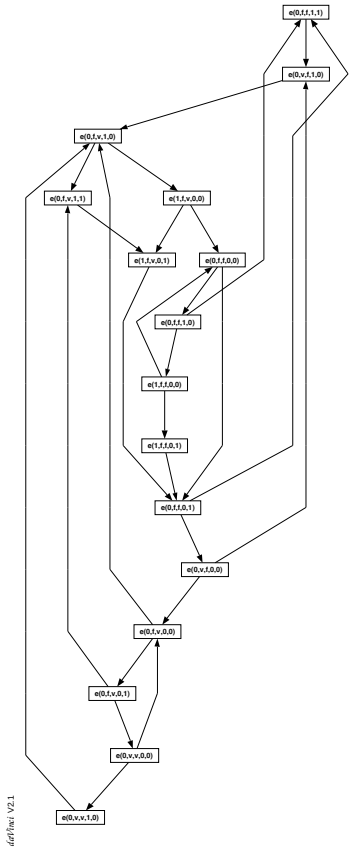

Figura 7: Un lector y un escritor

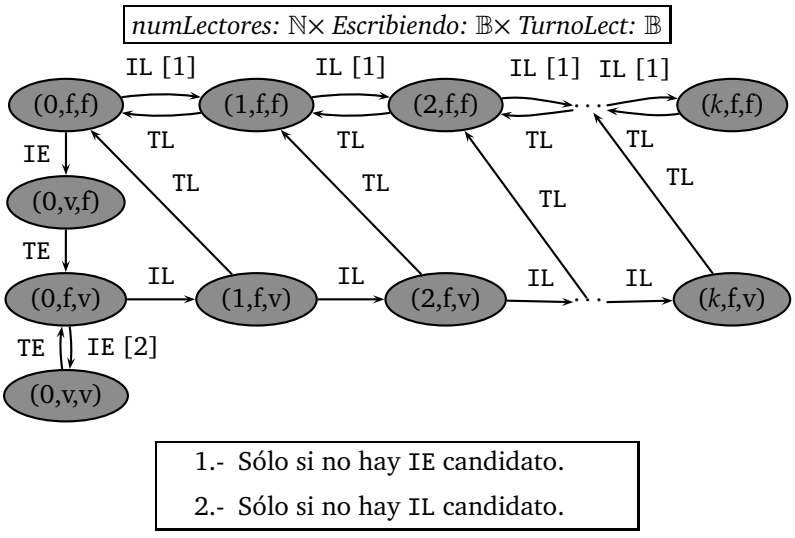

Figura 8: Grafo con anotaciones de determinismo (k lectores)

***CPRE*:** *Cierto*
CPRE: Cierto
Ingreso (b, s)
***POST*:** *Incrementamos el saldo*
POST: $b.Saldo^{sal} = b.Saldo^{ent} + s$

***CPRE*:** *Hay saldo suficiente*
CPRE: $b.Saldo \geq s$
Reintegro (b, s)
***POST*:** *Se decrementa el saldo*
POST: $b.Saldo^{sal} = b.Saldo^{ent} - s$

Como se puede observar, la CPRE de la operación *Reintegro* depende de uno de los parámetros de entrada, pero aplicar el esquema de desglose de operaciones visto hasta ahora puede no ser factible.

En efecto, si desglosamos la CPRE por posibles valores de la cantidad que se intenta sacar, obtendríamos un conjunto de entradas demasiado grande como para realizar replicación de código.

La solución que se suele adoptar en estos casos es la de desglosar las entradas por *procesos llamantes*, si el número de clientes de un objeto protegido es relativamente pequeño. El objeto protegido resultante tendrá una serie de propiedades especiales, pues en cada guarda de una entrada privada habrá a lo sumo un proceso en espera, hecho que puede ser utilizado a la hora de codificar.

Estructura del programa

Tendremos dos tipos de procesos, que llamaremos *padres* (que ingresan dinero) e *hijos* (que lo gastan). La operación *Reintegro* del recurso compartido habrá de tener un parámetro de entrada donde se especifica el número de proceso, que se le asigna en el momento de su creación:

```
type PID is Positive range 1..NumNenes;

MiCuenta : Cuenta

-- Los que ingresan y sacan dinero son tareas
-- Los padres ingresan dinero..
task type Papa_Type;
task body Papa_Type is
begin
   loop
      ...
      MiCuenta.Ingreso(500.0);
   end loop;
end Papa_Type;

-- ..y los hijos lo gastan
task type Nene_Type (Id : PID);
task body Nene_Type is
begin
   loop
      ...
      MiCuenta.Reintegro(Id, Peticion (Id));
   end loop;
end Nene_Type;
```

```
-- Lanzamos los procesos padres al declararlos
-- como componentes de un vector.
Papas : array(1..Num_Papas) of Papa_Type;

-- Los hijos los declaramos uno a uno
Nene1 : Nene_Type (1);
Nene2 : Nene_Type (2);
Nene3 : Nene_Type (3);
Nene4 : Nene_Type (4);
```

Desarrollo de la sincronización de seguridad

La traducción del recurso a objetos protegidos, toda vez que hemos decidido desglosar el reintegro de acuerdo con el identificador de proceso, es relativamente sencillo:

```
protected type Cuenta is
   entry Reintegro (Quien : PID; Cant : in Saldo_Type);
   entry Ingreso (Cant : in Saldo_Type);
private
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   Cantidad_Pendiente : Pendientes;
   -- para implementar las operaciones pendientes
   -- necesitamos una familia de entries
   entry Reintegro_Aplazado (PID)(Quien : PID; Cant : in Saldo_Type);
   -- ésta tiene que tener la misma cabecera de Reintegro
end Cuenta;

protected body Cuenta is
   entry Reintegro (Quien : in PID; Cant : in Saldo_Type)
   when True is
   begin
      Cantidad_Pendiente (Quien) := Cant;
      requeue Reintegro_Aplazado (Quien);
   end Reintegro;
   entry Ingreso(Cant : in Saldo_Type)
      -- siempre dejamos ingresar..
   when True is
   begin
      Saldo := Saldo + Cant;
   end Ingreso;
   entry Reintegro_Aplazado
      (for P in PID)
      (Quien : in PID; Cant : in Saldo_Type)
      -- es necesario que haya saldo para satisfacer la cantidad
      -- pendiente
      -- Cantidad_Pendiente (P) = Cant
   when Saldo >= Cantidad_Pendiente (P) is
   begin
      Saldo := Saldo - Cantidad_Pendiente (P);
      -- Cantidad_Pendiente ya no es válida
   end Reintegro_Aplazado;
end Cuenta;
```

Obsérvese que la familia de entradas varía en el rango de los identificadores de proceso, lo cual identifica únicamente un valor del saldo en espera. El vector *Cantidad_Pendiente* es necesario para poder guardar los parámetros que influyen en la CPRE.

Desarrollo de la sincronización de vivacidad

En su estado actual, esta solución adolece de riesgo de inanición — las peticiones de reintegro más elevadas podrían no ser atendidas nunca. Por ello, la especificación del problema debe completarse con ciertos criterios de vivacidad. Así, si lo prioritario es que no haya inanición, podríamos pasar a un esquema de atención por estricto orden de llegada.

En este caso, vamos a asumir que nos interesa *atender prioritariamente a las peticiones de mayor dinero*, independientemente de los problemas de inanición que puedan ocasionarse.

Esto se puede introducir en la solución anterior *fortaleciendo* la CPRE de *Reintegro* de la siguiente manera:

***CPRE*:** *$s \leq$ b.Saldo y no hay ninguna otra petición pendiente menor o igual que b.Saldo y mayor que s*
Reintegro (b, s)
POST: b.Saldo^{sal} = b.Saldo^{ent} - s

Obsérvese que esta CPRE informal ampliada no se puede codificar simplemente contando cuántos procesos están bloqueados en una entrada del objeto protegido.

Solución sin desbloqueos explícitos

La codificación directa de esta idea llevaría al siguiente código:

```
type TBloqueado is array (PID) of Boolean;
protected type Cuenta is
   entry Reintegro (Quien : PID; Cant : in Saldo_Type);
   entry Ingreso (Cant : in Saldo_Type);
private
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   Cantidad_Pendiente : Pendientes;
   Bloqueado : TBloqueado := (others => False);
   -- para implementar las operaciones pendientes
   -- necesitamos una familia de entries
   function NoHayMejorReintegro (Cant : Saldo_Type) return Boolean;
   entry Reintegro_Aplazado (PID)(Quien : PID; Cant : in Saldo_Type);
   -- ésta tiene que tener la misma cabecera de Reintegro
end Cuenta;

protected body Cuenta is
   entry Reintegro (Quien : in PID; Cant : in Saldo_Type)
   when True is
   begin
      Cantidad_Pendiente (Quien) := Cant;
      if Cant > Saldo then
         Bloqueado (Quien) := True;  -- Necesario para decidir mejor reintegro
      end if;
      requeue Reintegro_Aplazado (Quien);
   end Reintegro;
```

```
   entry Ingreso (Cant : in Saldo_Type)
      -- siempre dejamos ingresar..
   when True is
   begin
      Saldo := Saldo + Cant;
   end Ingreso;

   function NoHayMejorReintegro (Cant : Saldo_Type) return Boolean is
   begin
      for K in PID loop
         if Bloqueado (K) and then
            Saldo >= Cantidad_Pendiente (K) and then
            Cantidad_Pendiente (K) > Cant
         then return False;
         end if;
      end loop;
      return True;
   end NoHayMejorReintegro;

   entry Reintegro_Aplazado
      (for P in PID)
      (Quien : in PID; Cant : in Saldo_Type)
      -- es necesario que haya saldo para satisfacer la cantidad
      -- pendiente
   when Saldo >= Cantidad_Pendiente (P) and then
        NoHayMejorReintegro (Cantidad_Pendiente (P)) is
   begin
      Saldo := Saldo - Cantidad_Pendiente (P);
      -- Cantidad_Pendiente ya no es válida
      Bloqueado (P) := False;
   end Reintegro_Aplazado;
end Cuenta;
```

La traducción es bastante directa, encargándose la función auxiliar *NoHayMejorReintegro* de codificar la cuantificación existencial mencionada en el apartado anterior.

Sin embargo, podemos ver que si el número de clientes es alto, la reevaluación de las guardas puede ocasionar una cierta ineficiencia, motivo suficiente para pensar en una solución con desbloqueos explícitos.

Solución con desbloqueos explícitos

En esta solución el proceso que acaba de realizar un ingreso decide *en una sola pasada* qué proceso hijo es el siguiente en desbloquearse.

```
type Pendientes is array (PID) of Saldo_Type;
type TBloqueado is array (PID) of Boolean;
type Condition is (Ninguno, Reintegro);

protected type Cuenta is
   entry Reintegro (Quien : PID; Cant : in Saldo_Type);
   entry Ingreso (Cant : in Saldo_Type);
private
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
```

```
   Cantidad_Pendiente : Pendientes;
   Bloqueado : TBloqueado := (others => False);
   -- para implementar las operaciones pendientes
   -- necesitamos una familia de entries
   procedure BuscaReintegro;
   entry Reintegro_Aplazado (PID)(Quien : PID; Cant : in Saldo_Type);
   -- ésta tiene que tener la misma cabecera de Reintegro
   Siguiente : Condition := Ninguno;
   Cual : PID := 1; -- da igual
end Cuenta;

protected body Cuenta is
   entry Reintegro (Quien : in PID; Cant : in Saldo_Type)
   when True is
   begin
      Cantidad_Pendiente (Quien) := Cant;
      Bloqueado (Quien) := True; -- incondicionalmente
      BuscaReintegro;            -- se decide quién puede sacar dinero
      requeue Reintegro_Aplazado (Quien);
   end Reintegro;

   entry Ingreso (Cant : in Saldo_Type)
      -- siempre dejamos ingresar..
   when True is
   begin
      Saldo := Saldo + Cant;
      -- aquí se mete el desbloqueo explícito
      BuscaReintegro;
   end Ingreso;

   procedure BuscaReintegro is
      Mejor : PID;
      Encontrado : Boolean := False;
   begin
      Siguiente := Ninguno; -- redundante
      for K in PID loop
         if Bloqueado (K) and then
            Saldo >= Cantidad_Pendiente (K) and then
            (not Encontrado or else
             Cantidad_Pendiente (K) > Cantidad_Pendiente (Mejor))
         then
            Encontrado := True;
            Siguiente := Reintegro;
            Mejor := K;
         end if;
      end loop;
      if Siguiente = Reintegro
      then
         Cual := Mejor;
      end if;
   end BuscaReintegro;

   entry Reintegro_Aplazado
      (for P in PID)
```

```
      (Quien : in PID; Cant : in Saldo_Type)
      -- es necesario que haya saldo para satisfacer la cantidad
      -- pendiente
   when Saldo >= Cantidad_Pendiente (P) and then
         (Siguiente = Reintegro and then Cual = P) is
   begin
      Siguiente := Ninguno;
      Saldo := Saldo - Cantidad_Pendiente (P);
      -- Cantidad_Pendiente ya no es válida
      Bloqueado (P) := False;
      -- Desbloqueo de salida en cascada
      BuscaReintegro;
   end Reintegro_Aplazado;
end Cuenta;
```

Ahora el procedimiento *BuscaReintegro* es el encargado de ceder el turno a algún proceso bloqueado *si existe*.

La variable *Siguiente*, junto con el valor de *Cual* determinan únicamente un proceso a despertar. Los valores especiales *Ninguno* y *Cualquiera* son, muchas veces, redundantes, pero sirven para entender mejor el código de desbloqueos, la inicialización de *Siguiente*, etc.

Obsérvese también que no sólo los que ingresan invocan a *BuscaReintegro*, sino también los procesos que acaban de sacar dinero, pues puede ocurrir que un ingreso sea suficientemente elevado como para desbloquear a varios reintegros aplazados. Este mecanismo en que una secuencia de desbloqueos explícitos se implementa como un paso de testigo entre varias operaciones aplazadas se denomina tradicionalmente *despertar en cascada*.

Finalmente, cabe señalar que la mayor eficiencia se ha conseguido a costa de un código más oscuro y dificil de depurar.

Solución con atención por estricto orden de llegada

Finalizamos esta sección comentando la ya mencionada solución basada en atender las peticiones de reintegro por orden de llegada, independientemente de su cuantía. Es decir, si hay un reintegro pendiente por falta de saldo no se atienden otros reintegros, aunque estos sean de menor cuantía. Es, pues, una solución drástica, ya que reducimos notablemente la concurrencia del sistema. Se aplicaría entonces el esquema de la sección 3.4.2, notable por su sencillez:

```
protected type Cuenta is
   entry Reintegro (Cant : in Saldo_Type);
   entry Ingreso (Cant : in Saldo_Type);
private
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   Reintegro_Pendiente : Boolean := False;
   Cantidad_Pendiente : Saldo_Type;
   -- para implementar las operaciones pendientes
   entry Reintegro_Aplazado (Cant : in Saldo_Type);
   -- ésta tiene que tener la misma cabecera de Reintegro
end Cuenta;

protected body Cuenta is
   entry Reintegro (Cant : in Saldo_Type)
     -- no se puede sacar dinero si hay reintegros pendientes
   when not Reintegro_Pendiente is
   begin
```

```
      if Saldo > Cant then
         -- realizamos el reintegro
         Saldo := Saldo - Cant;
         -- dar recibo, etc
      else
         -- dejamos pendiente esta operación
         Reintegro_Pendiente := True;
         Cantidad_Pendiente := Cant;
         requeue Reintegro_Aplazado;
      end if;
   end Reintegro;

   entry Ingreso(Cant : in Saldo_Type)
     -- siempre dejamos ingresar..
   when True is
   begin
      Saldo := Saldo + Cant;
   end Ingreso;

   entry Reintegro_Aplazado (Cant : in Saldo_Type)
     -- es necesario que haya saldo para satisfacer la cantidad
     -- pendiente
   when Saldo >= Cantidad_Pendiente is
   begin
      Saldo := Saldo - Cantidad_Pendiente;
      Reintegro_Pendiente := False;
      -- Cantidad_Pendiente ya no es válida
   end Reintegro_Aplazado;
end Cuenta;
```

Referencias

[And91] Gregory R. Andrews. *Concurrent Programming. Principles and Practice*. The Benjamin/Cummnings Publishing Company. Inc., 1991.

[AS89] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. In N. Gehani and A.D. McGettrick, editors, *Concurrent Programming*. Addison-Wesley, 1989.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[Bri75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Software Engineering*, SE-1(2):199–207, 1975.

[BW98] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 1998.

[Coh95] Norman H. Cohen. *Ada as a Second Language*. McGraw-Hill, 1995.

[Hoa74] C.A.R. Hoare. Monitors, an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[TDBE01] T.S. Taft, R.A. Duff, R.L. Brukardt, and E.Ploedereder, editors. *Consolidated Ada Reference Manual. Language and Standard Libraries International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*. Springer Verlag, 2001.