# Especificación de recursos para programación concurrente

**Unidad de Programación**

**LSIIS**

**2005-11-15 23:08:28Z**

## Índice

# 1. Razones para especificar

La especificación de programas [vL00] es un amplio campo de trabajo en el que se utilizan muchas técnicas diferentes; entre las más conocidas se encuentran las basadas en estados [Abr96, Jon95, Spi92, Dil94] y variaciones de las mismas con orientación a objetos [Lan95]. La mayor parte de este trabajo se ha dedicado a la programación secuencial, como paso inicial necesario, aunque existen también propuestas dedicadas especialmente a la programación concurrente [Geh93, vLS79, Lam94]. Algunas de éstas pueden verse como aditamentos a técnicas diseñadas para la programación secuencial.

Entre las razones para utilizar especificación de programas en lugar de trabajar directamente en una implementación son:

**Formalismo:** un buen lenguaje de especificación es formal, lo que implica el ser no ambiguo y dar un método para obtener una interpretación única a partir de cada sentencia. Esta precisión permite expresar y transmitir ideas de forma inequívoca, algo indudablemente ventajoso. Hay que resaltar que un lenguaje formal es siempre preciso, pero lo contrario no es cierto. Adicionalmente, un lenguaje formal tiene una *teoría de la demostración* asociada. Ésta permite realizar razonamientos seguros acerca de sentencias escritas en ese lenguaje, y extraer conclusiones o probar propiedades sobre un sistema a partir de una descripción del mismo.

**Independencia del lenguaje de implementación:** la especificación es, idealmente, independiente del lenguaje de programación final. Esta independencia, junto con el nivel de expresividad de un lenguaje de especificación, idealmente más alto y menos necesitado de detalles que un lenguaje de programación, permiten plasmar de modo preciso qué es necesario programar utilizando una *lingua franca*. Adicionalmente, en muchos casos es posible transcribir de forma sencilla (y correcta) la especificación a un lenguaje de programación.

**Claridad y brevedad:** una buena especificación es fácil de entender y, muy a menudo, breve. Estas cualidades dependen tanto del lenguaje de especificación como del buen *arte* del especificador. La claridad y brevedad ayudan a comprender rápidamente una especificación y *ver* intuitivamente que es (o no) la deseada. Si la especificación de un problema es difícil de entender, falla en una parte muy importante de su objetivo.

**Demostrabilidad:** un lenguaje formal permite demostrar que una especificación cumple determinadas propiedades deseables. Una ventaja de hacerlo a este nivel es que es habitualmente más fácil que en la implementación en sí y que, una vez probadas, dichas propiedades quedan establecidas para cualquier implementación que respete la semántica de la especificación.

Es especialmente relevante el poder demostrar que un programa y una especificación se corresponden, ya sea con un método *ad-hoc* o mediante una derivación establecida de antemano. Ello permite establecer la corrección del programa a partir de la corrección de la especificación. Por ello es importante que la especificación sea clara (con objeto de manejarla con sencillez), formal (para poder razonar sobre ella con seguridad) e independiente del lenguaje (para poder aplicarla en cualquier caso).

Hay un compromiso entre la potencia del lenguaje de especificación y la demostrabilidad de teoremas en el mismo. En general, cuanto más expresivo es un lenguaje (formal), más difícil es el elaborar demostraciones en él. Incluso en sistemas tan simples como la lógica de primer orden más axiomas para modelizar la aritmética de los naturales existen teoremas verdaderos que son indemostrables [Göd80, Smu89, BA93].[1] La lógica de orden cero (proposicional) no tiene este problema, pero su expresividad es demasiado limitada para la mayor parte de los propósitos prácticos. Por otro lado, la sintaxis y propiedades de la lógica de primer orden son ampliamente

[1] Nótese que nuestro lenguaje formal se incluye en este caso, pues une a la lógica el uso implícito de la aritmética.

conocidas, lo que hace de ésta una elección clara como base de un lenguaje de especificación formal, aun con sus limitaciones de expresividad en un extremo y de semidecidibilidad en otro.

Cabe reseñar que los lenguajes de programación son[2] también lenguajes formales, pues hay un modelo (la máquina abstracta del lenguaje de programación) en la cual cualquier orden del lenguaje de programación tiene un comportamiento perfectamente definido. Sin embargo, la complejidad inherente de dicha máquina virtual hace que a menudo la demostración de cualquier propiedad interesante sea innecesariamente difícil.

# 2. Especificación de recursos

La especificación de recursos para programación concurrente que usaremos es una derivación de la utilizada para tipos abstractos de datos en ED-I y ED-II, ampliada sintáctica y semánticamente para incluir aspectos relativos a la sincronización de tareas. En particular, se pretende especificar claramente y por separado los dos aspectos de la sincronización en el acceso a un recurso compartido: la *exclusión mutua* y la *sincronización condicional*.

Un esquema mínimo de especificación sería:

**C-TADSOL** Nombre Recurso

**OPERACIONES**
**ACCIÓN** Operación_Recurso: *Tipo_Recurso[es]* $\times$ *Tipo*$_1$*[e]* $\times \ldots \times$ *Tipo*$_n$*[s]*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Recurso* $= \ldots$
**DONDE:** Tipo_Adicional $= \ldots$
**INVARIANTE:** $\ldots$

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

***CPRE*: *Precondición de concurrencia***
**CPRE:** $P(r, a_1, \ldots, a_n)$
**Operación_Recurso**$_1$**(r, a**$_1$**, . . . , a**$_n$**)**
***POST*: *Postcondición de la operación***
**POST:** $Q(r, a_1, \ldots, a_n)$

Existen ciertas similitudes y diferencias entre la especificación de un recurso compartido y de un tipo abstracto de datos (ver tabla 1). El uso de una derivación de una notación funcional de tipos abstractos de datos en lugar de, por ejemplo, una notación orientada a objetos, tiene como ventaja el poder reutilizar la mayor parte de la notación ya conocida. Como desventaja resaltable, la notación orientada a objetos es posiblemente la más natural para representar cambios en un recurso compartido; por ello muchos lenguajes de programación y notaciones para programación concurrente [AS89, Hoa74] utilizan conceptos de orientación a objetos. Es, sin embargo, posible pasar de una notación basada en funciones/acciones a una basada en objetos sin demasiados problemas y viceversa.

## 2.1. Declaración de nombre y de operaciones

La sección inicial de la especificación recoge el nombre del recurso, la declaración del nombre de cada operación y los tipos de sus argumentos:

[2] Deberían serlo, al menos, para permitir la compilación y/o interpretación por otro programa.

| **Similitudes** | **Diferencias** |
|---|---|
| ▪ Permite encapsulamiento de datos.<br>▪ Permite reusabilidad de abstracciones ya definidas.<br>▪ Es posible probar propiedades del recurso / TAD independientemente de cómo sea usado.<br>▪ Es posible razonar acerca de un proceso que utilice el recurso / TAD ateniéndose sólo a la especificación del mismo. | ▪ El estado del recurso depende del mundo externo. Por ejemplo, es necesario saber qué procesos usan un recurso en un momento dado y el estado de los mismos.<br>▪ Como consecuencia de lo anterior, no es posible tener un concepto de igualdad basado exclusivamente en un modelo del estado de los datos.<br>▪ De modo similar, no hay posibilidad de copia de recursos compartidos.<br>▪ Determinados argumentos de las operaciones, correspondientes al recurso, no guardan transparencia referencial: los procesos suspendidos en una operación ven (de forma implícita en el recurso) los cambios en el recurso compartido. |

Cuadro 1: Algunas similitudes y diferencias entre TADs y recursos compartidos

**ACCIÓN** Operación_Recurso$_1$: *Tipo_Recurso[es]* $\times$ *Tipo*$_1$*[e]* $\times \ldots \times$ *Tipo*$_n$*[s]*

Las operaciones declaradas en el interfaz son **el único medio de acceder al estado interno del recurso**, y lo realizan **en exclusión mutua**.[3] Cada operación está declarada como una acción para señalar que puede realizar un cambio en el valor de sus argumentos. Esto es especialmente importante al definir un recurso compartido, ya que muchas de las operaciones llevarán a cabo un cambio en el estado de dicho recurso.

La declaración de cada operación incluye:

- El nombre de la operación.
- La declaración del tipo de cada argumento. Como regla, el primer argumento en cada operación corresponderá al recurso compartido que se está especificando. Este argumento no aparecería en una notación orientada a objetos, pues las reglas sintácticas de ámbito de los identificadores permitirían acceder directamente al estado del objeto.
- La declaración del *modo* de cada argumento. Aunque es redundante, dado que se puede deducir de las postcondiciones de las operaciones, ayuda a una mejor comprensión del código. Un modo *[e]* (entrada) expresa que el parámetro no se modifica en la operación; *[s]* (salida) expresa que el valor del parámetro no se consulta, sino que se genera; y *[es]* expresa que el parámetro se consulta y su valor se cambia.

## 2.2. Declaración de dominio

El dominio del recurso define los tipos de los datos necesarios para representar el estado del recurso, así como ciertas restricciones (la invariante) en los valores del estado que no se pueden recoger en el lenguaje de tipos. El tipo del recurso debe contener la información necesaria relativa a la sincronización entre operaciones.

[3] Veremos más adelante (sección 2.6) cómo tratar los casos en los que se requiere exclusión parcial.

| Tipo | Declaración |
|---|---|
| Booleano | $\mathbb{B}$ |
| Natural | $\mathbb{N}$ |
| Enteros | $\mathbb{Z}$ |
| Real | $\mathbb{R}$ |

Cuadro 2: Tipos básicos

| Construcción | Declaración |
|---|---|
| Subrango | *A . . . B* |
| Enumeración | *a* \| *b* \| *c* |
| Unión | *nil* \| *Cons(Tipo_Dato, Tipo_Lista)* |
| Tuplas | *(Tipo*$_1 \times \ldots \times$ *Tipo*$_n$*)* |
| Constructor | *Nombre(Tipo*$_1 \times \ldots \times$ *Tipo*$_n$*)* |
| Conjunto | Conjunto*(Tipo_Dato)* |
| Secuencia | Secuencia*(Tipo_Dato)* |
| Función parcial | Tipo$_1 \nrightarrow$ Tipo$_2$ |

Cuadro 3: Constructores de tipos

La declaración de tipo puede usar ciertos tipos básicos (booleanos, naturales, enteros y reales) así como realizar la construcción de tipos derivados a partir de otros utilizando: unión de tipos (que incluye la enumeración), producto cartesiano (utilizando tuplas y constructores), conjuntos de datos de un tipo (serie no ordenada ni indexada), secuencias de datos de un tipo (serie ordenada e indexada) y funciones parciales (tablas). En el cuadro 2 se resumen los tipos básicos, junto con la sintaxis de su declaración, y en el cuadro 3 las construcciones de nuevos tipos. Es importante ver que el tipo booleano puede verse, perfectamente, como

**TIPO:** $\mathbb{B}$= *cierto* | *falso*

pero se incluye como tipo básico porque tiene una interpretación predefinida en la lógica de primer orden. Del mismo modo, $\mathbb{N}$ es un subconjunto de $\mathbb{Z}$ y se añade por comodidad. Asimismo la enumeración es un caso particular de la unión de tipos, y las tuplas y los constructores tienen la misma potencia. El conjunto, la secuencia y las funciones parciales pueden definirse a partir de las construcciones anteriores, pero se añaden, junto con sintaxis específica para manejar datos de esos tipos, por su utilidad.

La invariante permite expresar restricciones sobre el universo de valores descrito por el tipo, y dicha restricción debe cumplirse antes de la ejecución de cualquier operación, y al final de la misma (por tanto, la postcondición debe asegurarla, y ninguna precondición debería contradecirla). Puede violarse en el transcurso de la ejecución de una operación concurrente.

Por ejemplo, los números irracionales pueden ser descritos mediante el dominio:

**TIPO:** *Tipo_Irracional* $= \mathbb{R}$
**INVARIANTE:** $\forall i \in Tipo\_Irracional \bullet \nexists p, q \in \mathbb{Z} \bullet i = \frac{p}{q}$

El tipo de las secuencias de naturales estrictamente crecientes puede ser descrita mediante:

**TIPO:** *Tipo_Creciente* $=$ Secuencia*(*$\mathbb{N}$*)*
**INVARIANTE:** $\forall s \in Tipo\_Creciente \bullet (l = \text{Longitud}(s) \wedge (l < 2 \vee \forall k, 1 \leq k \leq l-1 \bullet s(k) < s(k+1)))$

## 2.3. Estado inicial del recurso

Cada recurso tiene un estado inicial que es necesario especificar. El estado inicial podría ser asignado mediante una operación al efecto (al estilo de un **Crear** de estructuras de datos), pero habitualmente la inicialización se produce sólo una vez y antes de cualquier otro acceso al recurso compartido. Esta restricción en su "puesta a cero" viene dada por el carácter del estado real del recurso compartido, que incluye no sólo los datos, sino el estado de las tareas que están accediendo a él. Adicionalmente, en muchos lenguajes es posible dar valores iniciales a un recurso sin necesidad de realizar una llamada explícita a una operación. En los que no es así, será necesario convertir la cláusula **Inicial** en el cuerpo de una operación a ser llamada al principio de la ejecución.

Para representar los valores iniciales del recurso se incluye una cláusula específica:

**INICIAL(r):** $I(r)$

La necesidad de validez de la fórmula $I(r)$ obliga a que el estado del recurso tome ciertos valores, que serán los iniciales. La variable $r$ tiene, de modo implícito, el tipo del recurso—el mismo tipo del primer argumento de cada una de las operaciones del recurso.

## 2.4. Sintaxis especial

Las fórmulas de la invariante, inicialización, precondiciones y postcondiciones siguen el lenguaje habitual de la lógica de primer orden. Se añaden las operaciones aritméticas habituales y un reducido número de facilidades para el acceso a los tipos de datos no básicos y para expresar nociones estrechamente relacionadas con la idea de programación.

En el caso de los tipos no básicos hemos tratado de ajustarnos a notaciones más o menos establecidas, como la del *mathematical toolkit* de Z [Spi92].

### 2.4.1. Tuplas con nombre y constructores con nombre

Se permite (pero no es obligatorio) nombrar algunos o todos los componentes de una tupla o constructor, de modo similar a como sucede con registros en un lenguaje procedimental. Eso permite referirse a dichos campos mediante su nombre: dado el tipo

**TIPO:** *Tipo_Dato = (a:*$\mathbb{R} \times$ *b:*$\mathbb{N} \times$ *c:*Secuencia*(*$\mathbb{Z}$*))*

una posible inicialización que utiliza algunos de los nombres de campos podría ser

**INICIAL(r):** $r.a = 0 \wedge \mathsf{Longitud}(r.c) = 0$

y otra, equivalente, utilizando igualdad[4] sería

**INICIAL(r):** $r = (a, \_, c) \wedge a = 0 \wedge \mathsf{Longitud}(c) = 0$

Las ventajas que ofrece el nombrado de campos es hacer la especificación algo más legible y el evitar tener que referirnos a campos que no van a ser alterados (en conjunción con la sintaxis mostrada en la sección 2.4.6).

### 2.4.2. Valores de entrada y salida

Nos será necesario consultar el estado de algunos argumentos de las operaciones a la entrada de la operación, y variar el mismo a la salida (en particular, lo necesitaremos para el recurso). Los superíndices $^{ent}$ y $^{sal}$ permiten diferenciar ambos estados.

### 2.4.3. Sintaxis y operaciones de secuencias

Las secuencias no tienen longitud fija y son indexables, representando un superconjunto de las listas y los vectores. El cuadro 4 recoge un resumen de la sintaxis de su uso, suponiendo que $r, s$ y $t$ son secuencias.

### 2.4.4. Sintaxis y operaciones para conjuntos

Las operaciones habituales de pertenencia, unión, intersección, diferencia, diferencia simétrica, complemento con respecto al universal, cardinalidad. En el cuadro 5 hay una selección de dichas operaciones con su significado.

[4] Obsérvese que permitimos una cuantificación existencial implícita sobre todas las variables libres de la fórmula.

| Sintaxis | Significado |
|---|---|
| Secuencia$(Tipo)$ | Declaración de tipo |
| $n =$ Longitud$(s)$ | $n$ es el numero de elementos en $s$, $0 \leq n$ |
| $s(n)$ | Elemento $n$-ésimo de la secuencia $s$, $0 < n \leq$ Longitud$(s)$ |
| $\langle\rangle$ | Secuencia vacía (Longitud$(\langle\rangle) = 0$) |
| $\langle a_1, a_2, \ldots, a_n\rangle$ | Secuencia con elementos: $\langle a_1 \ldots a_n\rangle(k) = a_k, 1 \leq k \leq n$ |
| $r = s(m..n)$ | Subsecuencia: $\forall i, m \leq i \leq n \bullet r(i - m + 1) = s(i)$ |
| $r = s + t$ | Concatenación: $ls =$ Longitud$(s) \land lt =$ Longitud$(t) \land$ Longitud$(r) = ls + lt \land$ $(\forall i, 1 \leq i \leq ls \bullet r(i) = s(i)) \land (\forall i, 1 \leq i \leq lt \bullet r(i + ls) = t(i))$ |

Cuadro 4: Sintaxis para secuencias

| Nombre | Definición |
|---|---|
| Unión | $A \cup B = \{x \mid x \in A \lor x \in B\}$ |
| Intersección | $A \cap B = \{x \mid x \in A \land x \in B\}$ |
| Diferencia | $A \setminus B = \{x \mid x \in A \land x \notin B\}$ |
| Diferencia simétrica | $A \triangle B = (A \cup B) \setminus (A \cap B)$ |

Cuadro 5: Operaciones básicas de conjuntos

### 2.4.5. Sintaxis y operaciones para funciones parciales

El tipo de las funciones parciales entre $A$ y $B$ se denota $A \nrightarrow B$ y se usa para formalizar la noción de tabla. Toda función parcial es una relación, es decir $A \nrightarrow B \subset$ Conjunto$(A \times B)$, por lo que se dispone de todas las operaciones sobre conjuntos de pares a la hora de manejar funciones parciales. Cuando nos referimos a funciones parciales, el par $(\mathit{clave}, \mathit{info})$ suele representarse $\mathit{clave} \mapsto \mathit{info}$.

El cuadro 6 resume algunas de las operaciones más frecuentes — usamos $T$ para referirnos a una tabla y $S$ para referirnos a un conjunto de claves.

| Nombre | Definición |
|---|---|
| Información asociada a una clave | $T(\mathit{clave})$ |
| Añadir una entrada nueva | $T \cup \{\mathit{clave} \mapsto \mathit{info}\} \quad (\neg \exists i \bullet \mathit{clave} \mapsto i \in T)$ |
| Modificar una entrada | $T \oplus \{\mathit{clave} \mapsto \mathit{info}\} = T \setminus \{\mathit{clave} \mapsto T(\mathit{clave})\} \cup \{\mathit{clave} \mapsto \mathit{info}\}$ |
| Borrar entradas por clave | $\{c_1 \ldots c_n\} \lhd T = \{(c, i) \mid (c, i) \in T \land c \notin \{c_1 \ldots c_n\}\}$ |
| Restricción de una tabla | $S \lhd T = \{(c, i) \mid (c, i) \in T \land c \in S\}$ |

Cuadro 6: Operaciones básicas con funciones parciales

### 2.4.6. Omisión de valores que no cambian

Se ha adoptado el operador " $\setminus$ " para denotar que los valores cuyo cambio no se hace explícito no se modifican durante el transcurso de una operación. En lugar de una fórmula como

**POST:** $r^{sal}.a = r^{ent}.a + 1 \land r^{sal}.b = r^{ent}.b \land r^{sal}.c = r^{ent}.c$

se podría escribir

**POST:** $r^{sal} = r \setminus r^{sal}.a = r^{ent}.a + 1$

En general admitiremos expresiones de la forma $r^{sal} = r \setminus s_1 = e_1 \wedge \cdots \wedge s_n = e_n$, asumiendo restricciones sintácticas que permiten su interpretación sin ambigüedad[5]. Esta sintaxis no es estrictamente necesaria, pero ayuda a abreviar y hacer más clara algunas especificaciones.

## 2.5. Especificación de operaciones

La semántica de cada operación incluye su precondición (fórmula que debe ser cierta inmediatamente antes de ejecutar la operación) y su postcondición (fórmula que debe ser cierta tras su ejecución y que determina el estado de las variables). Una llamada a una operación puede suspenderse si la precondición de concurrencia no se cumple. Esto hace que exista una diferencia notable entre el carácter de los argumentos de las operaciones: mientras que algunos de ellos son *propios* de cada operación (con un comportamiento idéntico al que se da en la programación secuencial), el que representa el recurso compartido es especial. Cuando una llamada se encuentra bloqueada, el estado del recurso compartido puede cambiar por efecto de otras llamadas cuyas precondiciones no han causado bloqueo. Este cambio no puede ocurrir con los argumentos que no se refieren a dicho recurso, pues son, en general, privados a la tarea que realizó la llamada e inaccesibles por las demás. Por otro lado, cuando una operación accede al recurso compartido, lo hace, como ya dijimos, en exclusión mutua.

La precondición puede descomponerse en aquellas fórmulas relacionadas con restricciones de uso de la operación (**PRE**) y fórmulas relacionadas únicamente con la especificación de las condiciones de concurrencia (**CPRE**). Las primeras no involucran al argumento correspondiente al recurso compartido, y las segundas lo involucran necesariamente. El no cumplimiento de una **CPRE** causa la suspensión de la llamada a la operación (algo completamente normal), mientras que el no cumplimiento de una **PRE** puede llevar a un error en tiempo de ejecución. Un ejemplo breve de especificación de una operación es:

**CPRE:** $s > 0$
**Wait(s)**
**POST:** $s^{sal} = s - 1$

Las precondiciones de concurrencia expresan condiciones de **seguridad**, cuyo cumplimiento es necesario para permitir el acceso al recurso compartido. No se reflejan, en principio, consideraciones de vivacidad: éstas requieren técnicas que dependen del lenguaje de programación en que se vaya a realizar la implementación final. En particular no se proporciona la capacidad de saber cuántos procesos están suspendidos en la condición de sincronización de una operación determinada, ni cuáles son. La razón de esta omisión es que determinadas propiedades no se pueden capturar de modo simple en lógica de primer orden, y, adicionalmente, en algunos mecanismos de concurrencia averiguar esto no es sencillo (o es, cuando menos, antinatural).

En la especificación de las precondiciones no se determina ningún orden de evaluación de las mismas. Asimismo, en las postcondiciones no se determina cómo se realiza el posible rearranque de las operaciones suspendidas, ni el orden de este rearranque. No hay consideración, en principio, sobre la carga de trabajo necesaria para la evaluación de dichas precondiciones o postcondiciones. Todos esos puntos se dejan a la implementación. Aunque se puede, evidentemente, intentar ser más puntilloso en la especificación, es preferible en esta fase que la claridad y corrección primen sobre consideraciones de eficiencia.

Se admiten también **PRE/CPRE/POST** en lenguaje natural con el propósito de **aclarar** las formales, no con el de sustituirlas. La especificación informal debe ser **informativa** y de **alto nivel**. Si la especificación informal no clarifica puntos que pueden quedar oscuros en la formal, su propósito se pierde y es preferible no adjuntarla o bien revisarla cuidadosamente. Como ejemplo véase la especificación del gestor de memoria (sección 3.7).

[5] y que omitimos para no hacer más tediosa la presentación.

## 2.6. Refinamiento de operaciones complejas

En muchos casos es necesario implementar la denominada *exclusión parcial*: determinadas operaciones realizadas a un recurso deben tener la posibilidad de solaparse en el tiempo (acceder sin exclusión mutua), mientras que otras deben excluirse. Aunque hemos dicho que supondremos exclusión mutua entre las operaciones de los recursos, es posible implementar exclusión parcial utilizando una técnica común: una operación **Op** se descompone en **Empezar_Op** y **Terminar_Op**. Entre ambas se realiza el acceso a los datos, guardados fuera del recurso, que se convierte así en un mero gestor de acceso, y no en un verdadero almacén de datos.

Con objeto de señalar este tipo de operaciones en una especificación inicial, y de documentarlas en una especificación más refinada, se añaden dos cláusulas adicionales a la especificación: **PROTOCOLOS** y **CONCURRENCIA**.

**CONCURRENCIA** describe qué combinaciones de operaciones pueden tener lugar concurrentemente. En una especificación inicial se referirá a acciones no directamente traspasables a una operación del C-TADSOL (pues requerirían no atomicidad). En una especificación refinada se refiere al nombre dado a una serie de operaciones mutuamente excluyentes, recogidas en la cláusula **PROTOCOLOS**.

**PROTOCOLOS** describe, en una especificación más refinada, cómo se realiza la descomposición de operaciones concurrentes en otras con exclusión mutua, que ya aparecen como operaciones en la especificación. Sólo es necesario recoger aquellas que se refieren al arbitrio del acceso, y no las que un hipotético cliente necesita para leer/cambiar datos que existen fuera del recurso gestor.

Como ejemplo de ambas cláusulas véase la especificación de *Lectores y Escritores* (sección 3.8) y *Paso de un puente* (sección 3.9).

El lenguaje de la cláusula de concurrencia incluye la ejecución concurrente ($\|$) de determinadas operaciones y la estrella de Kleene (*), que expresará aquí la posibilidad de ejecución **concurrente** de un número indeterminado de ocurrencias de lo repetido (incluidas cero repeticiones). Si no hay ninguna necesidad de especificar una ejecución concurrente puede obviarse o utilizarse el término *ninguna*, en lugar de una descripción de la concurrencia. Cada línea de la cláusula puede describirse mediante:

*línea_concurrencia* = *espec_concurrencia*
*espec_concurrencia* = **Op** |
(*espec_concurrencia*) |
*espec_concurrencia* $\|$ *espec_concurrencia* |
*espec_concurrencia**

donde **Op** es una operación de un recurso no refinado, o el nombre de una operación definida en la sección **PROTOCOLOS**, en un recurso ya más refinado.

El lenguaje de protocolos incluye secuenciación de operaciones (;) y la repetición (*) de operaciones, que expresará aquí la ejecución **secuencial** un número indeterminado de veces. Cada línea puede describirse como sigue:

*línea_protocolo* = Nombre_Op_Alto_Nivel: *espec_protocolo*
*espec_protocolo* = **Op** |
(*espec_protocolo*) |
*espec_protocolo* ; *espec_protocolo* |
*espec_protocolo**

donde **Op** puede ser cualquier operación del recurso.

La cláusula de **PROTOCOLOS** aclara el uso de las operaciones de la especificación desde el punto de vista de un cliente que las utiliza. La de **CONCURRENCIA** establece posibilidades de concurrencia que el recurso ha de respetar y poder implementar.

### 2.7. Referencia a otras fórmulas

En muchos casos resulta interesante referirse a otras fórmulas dentro de una dada. Aunque podrían replicarse las referenciadas, consideraciones de brevedad y mantenibilidad hacen aconsejable tener una sintaxis especial para este caso. La notación $\mathbf{CPRE}(Op_n(r, a_1 \ldots a_n))$ y $\mathbf{POST}(Op_n(r, a_1 \ldots a_n))$ permiten referirse a la precondición y postcondición de $Op_n$ con la instanciación de datos especificada. De igual modo, $\mathbf{Inicial}(r)$ permite referirse a la condición inicial del recurso. Se asume que en las **PRE/CPRE/POST** hay referencias implícitas a la invariante del recurso.

Esta facilidad puede utilizarse para referirse a precondiciones/postcondiciones de otros CTADs o TADs. La intención de las mismas es especificar una llamada a operaciones externas desde dentro del recurso, incluyendo los posibles efectos laterales de estas llamadas. En general debe usarse con precaución, pues a menudo no es buena idea efectuar llamadas a otros recursos desde dentro de uno dado.

### 2.8. Cláusulas de uso de otras abstracciones de datos

En caso de utilizar llamadas a otras abstracciones de datos es conveniente aclarar qué abstracción está siendo utilizada. La cláusula

**USA** Otro_TAD

tras el nombre del recurso siendo especificado permite poner esto de manifiesto.

## 3. Ejemplos de especificación

Veremos algunos ejemplos típicos de especificación. No son exhaustivos, pero sí suficientemente descriptivos. En ellos sólo se tratan las propiedades de seguridad, y no de vivacidad. En algunos casos sería posible añadir más datos conducentes a tener buenas propiedades de vivacidad, pero se ha preferido no hacerlo para tener una especificación lo más clara posible.

### 3.1. Control de entrada/salida de un aparcamiento

**C-TADSOL** Aparcamiento

**OPERACIONES**
**ACCIÓN** Entrar: *Tipo_Aparcamiento[es]*
**ACCIÓN** Salir: *Tipo_Aparcamiento[es]*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Aparcamiento = 0..Max*
**DONDE:** Max = ...
**INVARIANTE:** *cierto*

**INICIAL**(a)**:** $a = 0$

**CPRE:** *cierto*
**Salir(a)**
**POST:** $a^{sal} = a - 1$

**CPRE:** $a < Max$
**Entrar(a)**

**POST:** $a^{sal} = a^{ent} + 1$

## 3.2. Especificación de un semáforo

De manera especial, el semáforo exige una operación de inicialización entre sus operaciones. El uso racional de los semáforos incluye que esta operación se llame sólo una vez y al principio de la ejecución, antes de la llamada a cualquiera otra operación sobre el mismo semáforo. El estado inicial del semáforo queda indefinido en esta especificación. Esto **no** es lo habitual.

**C-TADSOL** Semáforo

**OPERACIONES**
**ACCIÓN** Init: *Tipo_Semáforo[s]* $\times \mathbb{N}$*[e]*
**ACCIÓN** Signal: *Tipo_Semáforo[es]*
**ACCIÓN** Wait: *Tipo_Semáforo[es]*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Semáforo* $= \mathbb{N}$
**INVARIANTE:** *cierto*
**INICIAL(**s**):** *cierto*

**CPRE:** *cierto*
**Init(s,v)**
**POST:** $s^{sal} = v$

**CPRE:** *cierto*
**Signal(s)**
**POST:** $s^{sal} = s^{ent} + 1$

**CPRE:** $s > 0$
**Wait(s)**
**POST:** $s^{sal} = s^{ent} - 1$

## 3.3. Almacén de un dato

**C-TADSOL** Almacén1Dato

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

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Almacén = (Dato: Tipo_Dato* $\times$ *HayDato:* $\mathbb{B}$*)*
**INVARIANTE:** *cierto*
**INICIAL(**b**):** $\neg b.HayDato$

**CPRE:** $b.HayDato$
**Tomar(b, e)**
**POST:** $e^{sal} = b^{ent}.Dato \wedge \neg b^{sal}.HayDato$

**CPRE:** $\neg b.HayDato$
**Poner(b, e)**
**POST:** $b^{sal}.Dato = e^{ent} \wedge b^{sal}.HayDato$

A partir de la especificación es posible comprobar que ambas operaciones son mutuamente excluyentes:

$$\begin{gathered}\mathbf{CPRE}(\mathbf{Tomar}(\mathbf{b}, \mathbf{e_1})) \wedge \mathbf{CPRE}(\mathbf{Poner}(\mathbf{b}, \mathbf{e_2})) \equiv \\ \neg b.HayDatos \wedge b.HayDatos \equiv \\ false\end{gathered}$$

En una implementación que respetase las condiciones de sincronización no sería necesario hacer explícita la exclusión mutua.

## 3.4. *Buffer* concurrente

**C-TADSOL** Buffer

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

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Buffer* = Secuencia*(Tipo_Dato)*
**INVARIANTE:** $\forall b \in Tipo\_Buffer$ • Longitud$(b) \leq MAX$
**DONDE:** $MAX = ...$
**INICIAL(**b**):** Longitud$(b) = 0$

***CPRE*: *El buffer no está vacío***
**CPRE:** Longitud$(b) > 0$
**Tomar(b, d)**
***POST*: *Retiramos un elemento del buffer***
**POST:** $l =$ Longitud$(b^{ent}) \wedge b^{ent}(1) = d^{sal} \wedge b^{sal} = b^{ent}(2..l)$

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

## 3.5. Buffer de pares e impares

La operación de extracción permite especificar si queremos retirar un número par o un número impar. Es posible (y es una técnica de implementación no inusual) desdoblar la operación **Tomar** en dos diferentes, una por cada valor del parámetro **t**.

**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* = Secuencia*(Tipo_Dato)*
*Tipo_Paridad = par|impar*
*Tipo_Dato* = $\mathbb{N}$
**INVARIANTE:** $\forall b \in Tipo\_Buffer\_PI \bullet \text{Longitud}(b) \leq MAX$
**DONDE:** $MAX = ...$
**INICIAL(**b**):** Longitud$(b) = 0$

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

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

## 3.6. Multibuffer

En este ejemplo, además de precondiciones de concurrencia, hay precondiciones de uso (**PRE**). Éstas no se utilizan para realizar sincronización; su violación causaría un comportamiento indefinido. Por otro lado, el manejo del estado del recurso compartido se ha realizado mediante operaciones de concatenación de secuencias. Se puede ver la diferencia con las de las secciones 3.4 y 3.5.

**C-TADSOL** MultiBuffer

**OPERACIONES**
**ACCIÓN** Poner: *Tipo_Multi_Buffer[es]* $\times$ *Tipo_Secuencia[e]*
**ACCIÓN** Tomar: *Tipo_Multi_Buffer[es]* $\times$ *Tipo_Secuencia[s]* $\times$ $\mathbb{N}$*[e]*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Multi_Buffer* = Secuencia*(Tipo_Dato)*
*Tipo_Secuencia = Tipo_Multi_Buffer*
**INVARIANTE:** $\forall b \in Tipo\_Multi\_Buffer \bullet \text{Longitud}(b) \leq MAX$
**DONDE:** $MAX = ...$
**INICIAL(**b**):** $b = \langle\rangle$

**PRE:** $n \leq \lfloor MAX/2 \rfloor$
***CPRE*:** *Hay suficientes elementos en el multibuffer*
**CPRE:** Longitud$(b) \geq n$
**Tomar(b, s, n)**
***POST*:** *Retiramos elementos*
**POST:** $n = \text{Longitud}(s^{sal}) \wedge b^{ent} = s^{sal} + b^{sal}$

**PRE:** Longitud$(s) \leq \lfloor MAX/2 \rfloor$

***CPRE*:** *Hay sitio en el buffer para dejar la secuencia*
**CPRE:** Longitud$(b + s) \leq MAX$
**Poner(b, s)**
***POST*:** *Añadimos una secuencia al buffer*
**POST:** $b^{sal} = b^{ent}$ + $s^{ent}$

## 3.7. Gestor de memoria

La solicitud de memoria devuelve una dirección $i$ a partir de la cual se encuentran disponibles $n$ posiciones libres de memoria. Por simplicidad de la implementación, suponemos que a la hora de devolver memoria, el cliente dice no sólo en qué lugar empieza la zona que le ha sido asignada, sino también la cantidad de memoria pedida.

**C-TADSOL** Gestor de memoria

**OPERACIONES**
**ACCIÓN** Solicitar: *Tipo_GM[es]* $\times$ *Tipo_Tamaño[e]* $\times$ *Tipo_Dirección[s]*
**ACCIÓN** Devolver: *Tipo_GM[es]* $\times$ *Tipo_Tamaño[e]* $\times$ *Tipo_Dirección[e]*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_GM = Tipo_Direccion* $\nrightarrow \mathbb{B}$
*Tipo_Tamaño = 1..Max*
*Tipo_Dirección = 1..Max*
**DONDE:** $Max = ...$
**INICIAL(**m**):** $\forall i, 1 \leq i \leq Max \bullet \neg m(i)$

***CPRE*:** *Hay* $n$ *posiciones consecutivas de memoria libre*
**CPRE:** $\exists k, 1 \leq k \leq Max - n + 1 \bullet (\forall j, k \leq j < k + n \bullet \neg m(j))$
[6] **Solicitar(m, n, i)**
***POST*:** *A partir de* $i$ *había* $n$ *posiciones de memoria libre que están ahora señaladas como ocupadas*
**POST:** $\forall j \in 0 \ldots n-1 \bullet \neg m^{ent}(i + j) \wedge m^{sal} = m^{ent} \oplus \{j \mapsto \underline{\mathrm{t}} | i \leq j < i + n\}$

***PRE*:** *Hay* $n$ *posiciones de memoria ocupada a partir de la* $i$*-ésima*
**PRE:** $\forall k, i \leq k < n + i \bullet m(k)$
**CPRE:** cierto
**Devolver(m, n, i)**
***POST*:** *Se marcan como libres las* $n$ *posiciones de memoria a partir de la* $i$*-ésima*
**POST:** $m^{sal} = m^{ent} \oplus \{j \mapsto \underline{\mathrm{f}} | i \leq j < i + n\}$

## 3.8. Lectores y escritores

Este es un caso típico de exclusión parcial. La especificación recogida abajo presupone exclusión total entre las operaciones, y recoge cómo se deben utilizar para implementar lecturas concurrentes y escrituras excluyentes. Un paquete que implementase de modo transparente las operaciones **Leer** y **Escribir** utilizaría un tipo similar a

**TIPO:** *Tipo_Base_Datos = (Tipo_Almacén_Datos* $\times$ *Tipo_Gestor_LE)*

[6] También podríamos haber escrito $m = f \cup \{j \mapsto \underline{\mathrm{f}} | k \leq j < k + n\}$

Incidentalmente, el método de acceso que implementa el ejemplo de lectores y escritores es el mismo que utiliza Ada en objetos protegidos que incluyen funciones.

**C-TADSOL** Lectores / Escritores

**OPERACIONES**
**ACCIÓN** Iniciar_Lectura: *Tipo_Gestor_LE[es]*
**ACCIÓN** Iniciar_Escritura: *Tipo_Gestor_LE[es]*
**ACCIÓN** Terminar_Lectura: *Tipo_Gestor_LE[es]*
**ACCIÓN** Terminar_Escritura: *Tipo_Gestor_LE[es]*

**PROTOCOLOS:** Leer: Iniciar_Lectura; Terminar_Lectura
Escribir: Iniciar_Escritura; Terminar_Escritura
**CONCURRENCIA:** Leer*

**SEMÁNTICA**
**DOMINIO:**
**TIPO:** *Tipo_Gestor_LE = (NLect:* $\mathbb{N}\times$ *Esc:* $\mathbb{B}$*)*
**INVARIANTE:** $\forall g \in Tipo\_Gestor\_LE \bullet (g.Esc \rightarrow g.NLect = 0)$
**INICIAL(g):** $\neg g.Esc \wedge g.NLect = 0$

**CPRE:** $\neg g.Esc$
**Iniciar_Lectura(g)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.NLect = 1 + g^{ent}.NLect$

**CPRE:** *cierto*
**Terminar_Lectura(g)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.NLect = g^{ent}.NLect - 1$

**CPRE:** $\neg g.Esc \wedge g.NLect = 0$
**Iniciar_Escritura(g)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.Esc$

**CPRE:** *cierto*
**Terminar_Escritura(g)**
**POST:** $g^{sal} = g^{ent} \setminus \neg g^{sal}.Esc$

## 3.9. Control del paso de un puente con peso máximo

Similar al anterior, incluye sincronización con respecto a los argumentos de entrada.

**C-TADSOL** Puente

**OPERACIONES**
**ACCIÓN** Entrar_NS: *Tipo_Gestor_Puente[es]* $\times$ $\mathbb{R}$*[e]*
**ACCIÓN** Salir_NS: *Tipo_Gestor_Puente[es]* $\times$ $\mathbb{R}$*[e]*
**ACCIÓN** Entrar_SN: *Tipo_Gestor_Puente[es]* $\times$ $\mathbb{R}$*[e]*
**ACCIÓN** Salir_SN: *Tipo_Gestor_Puente[es]* $\times$ $\mathbb{R}$*[e]*

**PROTOCOLOS:** Pasar_SN: Entrar_SN; Salir_SN
Pasar_NS: Entrar_NS; Salir_NS
**CONCURRENCIA:** Pasar_SN*
Pasar_NS*
**SEMÁNTICA**

**DOMINIO:**
**TIPO:** *Tipo_Gestor_Puente = (Cruzan_NS:* $\mathbb{N}\times$ *Cruzan_SN:* $\mathbb{N}\times$ *Peso_Restante:* $\mathbb{N}$*)*
**INVARIANTE:** $\forall g \in Tipo\_Gestor\_Puente \bullet (g.Cruzan\_SN = 0 \vee g.Cruzan\_NS = 0)$
**INICIAL(g):** $g.Cruzan\_NS = 0 \wedge g.Cruzan\_SN = 0 \wedge g.Peso\_Restante = Max$
**DONDE:** $Max = ...$

**PRE:** $p \leq Max$
**CPRE:** $g.Cruzan\_NS = 0 \wedge g.Peso\_Restante - p \geq 0$
**Entrar_SN(g, p)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.Cruzan\_SN = g^{ent}.Cruzan\_SN + 1 \wedge g^{sal}.Peso\_Restante = g^{ent}.Peso\_Restante - p$

**CPRE:** *cierto*
**Salir_SN(g, p)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.Cruzan\_SN = g^{ent}.Cruzan\_SN - 1 \wedge g^{sal}.Peso\_Restante = g^{ent}.Peso\_Restante + p$

**PRE:** $p \leq Max$
**CPRE:** $g.Cruzan\_SN = 0 \wedge g.Peso\_Restante - p \geq 0$
**Entrar_NS(g, p)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.Cruzan\_NS = g^{ent}.Cruzan\_NS + 1 \wedge g^{sal}.Peso\_Restante = g^{ent}.Peso\_Restante - p$

**CPRE:** *cierto*
**Salir_NS(g, p)**
**POST:** $g^{sal} = g^{ent} \setminus g^{sal}.Cruzan\_NS = g^{ent}.Cruzan\_NS - 1 \wedge g^{sal}.Peso\_Restante = g^{ent}.Peso\_Restante + p$

# Referencias


[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[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.

[BA93] M. Ben-Ari. *Mathematical Logic for Computer Science*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1993.

[Dil94] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994.

[Geh93] Narain H. Gehani. Capsules: a shared memory access mechanism for Concurrent C/ C++. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):795–811, 1993.

[Göd80] K. Gödel. *Sobre proposiciones formalmente indecidibles de los principia mathematica y sistemas afines*. Universidad de Valencia, 1980.

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

[Jon95] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1995.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lan95] K. Lano. *Formal Object-Oriented Development*. Springer-Verlag, 1995.

[Smu89] R. Smullyan. *Juegos para imitar a un pájaro imitador*. GEDISA, 1989.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000. Available at `http://lml.ls.fi.upm.es/babel/babylon/fsr-avl.pdf`.

[vLS79] A. van Lamsweerde and M. Sintzoff. Formal derivation of strongly correct concurrent programs. *Acta Informatica*, 12, 1979.