Especificación de recursos para programación concurrente

Manuel Carro Julio Mariño

Ángel Herranz

Dpto. de Lenguajes, Sistemas Informáticos e Ing. de Software
Universidad Politécnica de Madrid

Revisión SVN 792/ 2011-10-18

Índice

1. Razones para especificar **2**

2. Especificación de recursos **3**
2.1. Declaración de nombre y de operaciones 4
2.2. Declaración de dominio 5
2.3. Estado inicial del recurso 7
2.4. Lenguaje de fórmulas 7
2.4.1. Tuplas con nombre y constructores con nombre 7
2.4.2. Valores de entrada y salida 8
2.4.3. Sintaxis y operaciones de secuencias 8
2.4.4. Sintaxis y operaciones para conjuntos 8
2.4.5. Sintaxis y operaciones para funciones parciales 9
2.4.6. Omisión de valores que no cambian 9
2.5. Especificación de operaciones 9
2.6. Refinamiento de operaciones complejas 10
2.7. Referencia a otras fórmulas 12
2.8. Cláusulas de uso de otras abstracciones de datos 12

3. Ejemplos de especificación **12**
3.1. Control de entrada/salida de un aparcamiento 12
3.2. Especificación de un semáforo 13
3.3. Almacén de un dato 14
3.4. *Buffer* concurrente 15
3.5. Buffer de pares e impares 16
3.6. Multibuffer 16
3.7. Gestor de memoria 17
3.8. Lectores y escritores 18

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.

No es menos importante, especialmente en fases tempranas del desarrollo de una aplicación compleja o crítica, la capacidad de demostrar que una especificación sufre de unos determinados defectos. Esto facilita enormemente el desarrollo incremental de la especificación y permite iniciar la codificación con un grado de confianza mucho mayor.

Trazabilidad: Es especialmente relevante el poder demostrar que un programa y una especificación se corresponden, ya sea con un método *ad-hoc* o median-

te 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).

En el caso particular que nos ocupa, el de las especificaciones formales de sistemas concurrentes, los argumentos arriba mostrados hay que interpretarlos a la luz de las dificultades específicas de la concurrencia:

- El uso del lenguaje de especificación como lingua franca independiente del lenguaje de programación es especialmente relevante en aplicaciones concurrentes habida cuenta del desigual apoyo a la concurrencia que estos proporcionan, lo que dificulta enormemente razonar sobre estos aspectos. Además, el desarrollo de un sistema concurrente a partir de especificaciones formales facilita portarlas a otro lenguaje una vez que aquel en que fueron desarrolladas queda obsoleto.
- La formalidad, claridad y concisión son esenciales para permitir a los desarrolladores razonar sobre el comportamiento esperado del sistema, ya que, como hemos dicho, hacerlo directamente sobre el código fuente suele ser impracticable.
- Finalmente, debido al indeterminismo inherente a los sistemas concurrentes, que impide el uso de las técnicas de prueba como en los sistemas secuenciales, es de vital importancia la posibilidad de automatizar ya la demostración de propiedades o, si esto no es viable, al menos la predicción de defectos, quizá mediante generación sistemática de escenarios de ejecución.

La notación que vamos a presentar está pensada teniendo en especial consideración este último aspecto, existiendo experimentos que demuestran la capacidad de detección automática de problemas de vivacidad mediane la traducción de estas especificaciones a formalismos lógicos de más bajo nivel [HMCM09].

Por otra parte, la trazabilidad será asegurada *a priori*, es decir, se proporcionará un mecanismo semiautomático de construcción del código a partir de las especificaciones que obviará la necesidad de demostrar a posteriori que un determinado código es conforme a dicha especificación [CMAHM04].

2. Especificación de recursos

La notación para especificar recursos para programación concurrente que usaremos es una extensión de la utilizada para especificar tipos abstractos de datos (o clases) en asignaturas del anterior plan de estudios de Ingeniería Informática, 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-TAD Nombre Recurso

OPERACIONES
ACCIÓN Operación_Recurso: $Tipo_1$ *[e]* × ... × $Tipo_n$ *[s]*

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

INICIAL: *Fórmula sobre self que especifica el valor inicial del recurso*

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

Existen ciertas similitudes y diferencias entre la especificación de un recurso compartido y de un tipo abstracto de datos o clase (ver cuadro 1).

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:

ACCIÓN Operación_Recurso$_1$: $Tipo_1$ *[e]* × ... × $Tipo_n$ *[s]*

Las operaciones declaradas en la interfaz son *el único medio de acceder al estado interno del recurso*, y lo realizan *en exclusión mutua*.[1] 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.
- 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.

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

Similitudes	**Diferencias**
▪ Permite encapsulamiento de datos. ▪ Permite reusabilidad de abstracciones ya definidas. ▪ Es posible probar propiedades del recurso / TAD independientemente de cómo sea usado. ▪ 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. ▪ Como consecuencia de lo anterior, no es posible tener un concepto de igualdad basado exclusivamente en un modelo del estado de los datos. ▪ De modo similar, no hay posibilidad de copia de recursos compartidos. ▪ 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 (o clases) y recursos compartidos

- En esta versión de la notación, existe un parámetro oculto, como en cualquier lenguaje orientado a objetos. Dicho parámetro es accesible con la palabra self, su tipo es el tipo el dominio del recurso (ver siguiente apartado) y su modo es *[es]*.

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.

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

Cuadro 2: Tipos básicos

Construcción	Declaración
Subrango	$A \ldots B$
Enumeración	$a \mid b \mid c$
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	$\text{Tipo}_1 \nrightarrow \text{Tipo}_2$
Función total	$\text{Tipo}_1 \rightarrow \text{Tipo}_2$

Cuadro 3: Constructores de tipos

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: $\nexists\, p, q \in \mathbb{Z} \bullet \text{self} = \frac{p}{q}$

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

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

2.3. Estado inicial del recurso

Cada recurso tiene un estado inicial que es necesario especificar. Puesto que la inicialización se produce sólo una vez y antes de cualquier otro acceso al recurso compartido, se dispone de una cláusula especifica que establece las condiciones que debe cumplir el recurso antes de que ningún proceso invoque sus operaciones:

INICIAL: I(self)

La necesidad de validez de la fórmula I(self) restringe los valores que el recurso puede tomar inicialmente, si bien en general sólo nos interesa especificar completamente parte del estado inicial. Es decir, la cláusula de inicialización puede perfectamente no decir nada de parte del estado. La variable self tiene, de modo implícito, el tipo del recurso — el mismo tipo del argumento elidido de cada una de las operaciones del recurso.

Pensando en términos de orientación a objetos, podemos ver la cláusula de inicialización del recurso como una forma concisa de proporcionar la postcondición de un constructor estándar. En aquellos casos en que es necesario inicializar un recurso pasando algún argumento – como en un constructor alternativo – es más conveniente la declaración de constructores,[2] tal como se muestra en el ejemplo de la sección 3.2.

2.4. Lenguaje de fórmulas

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: $Tipo\_Dato = (a{:}\mathbb{R} \times b{:}\mathbb{N} \times c{:}\mathsf{Secuencia}(\mathbb{Z}))$

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

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

y otra, equivalente, utilizando igualdad[3] sería

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

[2] Esta característica se ha introducido en el curso 2010-2011.

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

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

Al especificar las postcondiciones de las operaciones del recurso suele ser necesario relacionar el estado del recurso inmediatamente antes a la invocación de la operación con el estado inmediatamente después de la ejecución completa de dicha operación. Algo similar ocurre al especificar el comportamiento de los parámetros con modo [*es*]. El superíndice ^{pre} nos permite diferenciar los valores de la variable en esos dos instantes.[4]

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.

Sintaxis	**Significado**
Secuencia(*Tipo*)	Declaración de tipo
$n = \mathsf{Longitud}(s)$	n es el numero de elementos en s, $0 \leq n$
$s(n)$	Elemento n-ésimo de la secuencia s, $0 < n \leq \mathsf{Longitud}(s)$
$\langle\rangle$	Secuencia vacía ($\mathsf{Longitud}(\langle\rangle) = 0$)
$\langle a_1, a_2, \ldots, a_n\rangle$	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: $ls = \mathsf{Longitud}(s) \wedge lt = \mathsf{Longitud}(t) \wedge \mathsf{Longitud}(r) = ls + lt \wedge$ $(\forall\, i, 1 \leq i \leq ls \bullet r(i) = s(i)) \wedge (\forall\, i, 1 \leq i \leq lt \bullet r(i+ls) = t(i))$

Cuadro 4: Sintaxis para 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.

Nombre	**Definición**
Unión	$A \cup B = \{x \mid x \in A \vee x \in B\}$
Intersección	$A \cap B = \{x \mid x \in A \wedge x \in B\}$
Diferencia	$A \setminus B = \{x \mid x \in A \wedge x \notin B\}$
Diferencia simétrica	$A \triangle B = (A \cup B) \setminus (A \cap B)$

Cuadro 5: Operaciones básicas de conjuntos

[4] Hasta el curso 2010-2011 hemos usado dos superíndices ^{ent} y ^{sal}. La notación actual está inspirada en la que usa el lenguaje de especificación JML (*Java modeling language*) [CKLP06].

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 $A \nrightarrow B \subset A \times B$, 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 (*clave*, *info*) suele representarse $clave \mapsto info$.

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

Cuadro 6: Operaciones básicas con funciones parciales

2.4.6. Omisión de valores que no cambian

Se ha adoptado el operador " \ " 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: self.a = self^{pre}.a + 1 $\wedge$ self.b = self^{pre}.b $\wedge$ self.c = self^{pre}.c

se podría escribir

POST: self = self \ self.a = self^{pre}.a + 1

En general admitiremos expresiones de la forma self = self \ $s_1 = e_1 \wedge \cdots \wedge s_n = e_n$, asumiendo restricciones sintácticas que permiten su interpretación sin ambigüedad.[5] Esta sintaxis no es estrictamente necesaria, pero puede ayudar a abreviar y hacer más clara algunas especificaciones. En caso de duda sobre cómo usar esta notación abreviada aconsejamos usar la "normal", aún a riesgo de escribir más.

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

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

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: self > 0
  Wait()
POST: self = self − 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).

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 ha propuesto añadir dos cláusulas adicionales a la especificación: **PROTOCOLOS** y **CONCURRENCIA**.

CONCURRENCIA describiría 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 describiría, 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.

En este momento, este tipo de extensiones de la notación está siendo *reconsiderada* y es posible que sean redefinidas o eliminadas. Por tanto, el resto de esta subsección es *totalmente prescindible de cara al curso actual*.

Como ejemplo de uso ambas cláusulas puede verse la especificación de *Lectores y Escritores* (sección 3.8) .

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 **CPRE**(self.$Op_n(a_1 \dots a_n)$) y **POST**(self.$Op_n(a_1 \dots a_n)$) permiten referirse a la precondición y postcondición de Op_n con la instanciación de datos especificada. De igual modo, **Inicial**(self) 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.

No obstante, debemos tener cuidado al incluir operaciones de otros TADs ya que estas pueden ser *parciales*, a diferencia de las presentes en el juego de herramientas matemático que hemos venido usando hasta ahora, que sólo comprende operaciones totales. Un ejemplo típico es referirse al primer elemento de una pila, que puede no estar definido en caso de estar dicha pila vacía. Para evitar estos problemas es necesario hacer uso de las fórmulas relativas a las PREs de dichas operaciones, tal como se explica en el apartado anterior.

Recordemos también que la especificación formal de las POSTs de las acciones de un recurso *no* permite hacer referencia a acciones de otros recursos.

3. Ejemplos de especificación

Veremos algunos ejemplos típicos de especificación de recursos. 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-TAD Aparcamiento

OPERACIONES
ACCIÓN Entrar: *(no poner nada indica que no tiene argumentos)*
ACCIÓN Salir:

SEMÁNTICA

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

INICIAL: self = 0

CPRE: *cierto*
Salir()
POST: self = self − 1

CPRE: self < *Max*
Entrar()
POST: self = self^{pre} + 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.

C-TAD Semáforo

OPERACIONES
ACCIÓN Init: ℕ*[e]*
ACCIÓN Signal:
ACCIÓN Wait:

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Semáforo = (Cont:*ℕ× *Listo:*𝔹*)*
INVARIANTE: *cierto*

INICIAL: ¬*self.Listo*

PRE: ¬self.*Listo*
Init(v)
POST: self.*Cont* = *v* ∧ self.*Listo*

PRE: self.*Listo*
CPRE: *cierto*
Signal()
POST: self.*Cont* = self^{pre}.*Cont* + 1 ∧ self.*Listo*

PRE: self.*Listo*
CPRE: self.*Cont* > 0

Wait()
POST: self.*Cont* = self^{pre}.*Cont* − 1 ∧ self.*Listo*

Para evitar el artefacto de la variable *Listo* en cualquier recurso que presente un protocolo de inicialización similar, admitimos ahora la posibilidad de usar constructores explícitos, que ya se asume que van a ser llamados una sola vez, no tienen componente de sincronización y su precondición, en caso de existir, sólo puede hacer referencia a los argumentos.

A continuación mostramos como queda este ejemplo usando constructores explícitos:

C-TAD Semáforo

OPERACIONES
CONSTRUCTOR Init: *Tipo_Semáforo* × ℕ
ACCIÓN Signal: *Tipo_Semáforo[es]*
ACCIÓN Wait: *Tipo_Semáforo[es]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Semáforo* = ℕ

PRE: —
Init(v)
POST: self= v

CPRE: *cierto*
Signal()
POST: self = self^{pre} + 1

CPRE: self > 0
Wait(s)
POST: self = self^{pre} − 1

3.3. Almacén de un dato

C-TAD Almacén1Dato

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

SEMÁNTICA
DOMINIO:
TIPO: *Almacén1Dato = (Dato: Tipo_Dato* × *HayDato:* 𝔹*)*
INVARIANTE: *cierto*

INICIAL: ¬self.*HayDato*

CPRE: self.*HayDato*
Tomar(e)
POST: $e = \text{self}^{pre}.Dato \wedge \neg\text{self}.HayDato$

CPRE: $\neg$self.*HayDato*
Poner(e)
POST: $\text{self}.Dato = e^{pre} \wedge \text{self}.HayDato$

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

$$\begin{aligned}&\textbf{CPRE}(\text{self}.\textbf{Tomar}(\mathbf{e_1})) \wedge \textbf{CPRE}(\text{self}.\textbf{Poner}(\mathbf{e_2})) \equiv \\ &\quad \neg\text{self}.HayDatos \wedge \text{self}.HayDatos \equiv \\ &\quad\quad falso\end{aligned}$$

En una implementación que respetase las condiciones de sincronización no sería necesario hacer explícita la exclusión mutua.[6]

3.4. *Buffer* concurrente

C-TAD Buffer

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

SEMÁNTICA
DOMINIO:
TIPO: *Buffer* = Secuencia *(Tipo_Dato)*
INVARIANTE: $\text{Longitud}(\text{self}) \leq MAX$

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

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

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

[6] Por eso este ejemplo es mucho más fácil de implementar con semáforos binarios que un almacén de n datos.

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-TAD BufferPI

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

SEMÁNTICA
DOMINIO:
TIPO: *Buffer_PI* = Secuencia *(Tipo_Dato)*
Tipo_Paridad = *par*|*impar*
Tipo_Dato = $\mathbb{N}$
INVARIANTE: Longitud(self) ≤ *MAX*

DONDE: *MAX* = ...
INICIAL: Longitud(self) = 0

***CPRE*:** *El buffer no está lleno*
CPRE: Longitud(self) < *MAX*
Poner(d)
***POST*:** *Añadimos un elemento al buffer*
POST: l = Longitud(self^{pre}) ∧ Longitud(self) = $l+1$ ∧ self($l+1$) = d^{pre} ∧ self(1..l) = self^{pre}
***CPRE*:** *El buffer no está vacío y el primer dato preparado para salir es del tipo que requerimos*
CPRE: Longitud(self) > 0 ∧ *Concuerda*(self(1), t)
DONDE: *Concuerda*(d, t) ≡ (d *mod* 2 = 0 ↔ t = *par*)
Tomar(d, t)
***POST*:** *Retiramos el primer elemento del buffer*
POST: l = Longitud(self^{pre}) ∧ self^{pre}(1) = d ∧ self = self^{pre}(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-TAD MultiBuffer

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

SEMÁNTICA
DOMINIO:
TIPO: *Multi_Buffer* = Secuencia *(Tipo_Dato)*
Tipo_Secuencia = Tipo_Multi_Buffer
INVARIANTE: Longitud(self) ≤ *MAX*

DONDE: *MAX* = ...
INICIAL: self = ⟨⟩

PRE: $n \leq \lfloor MAX/2 \rfloor$
***CPRE*:** *Hay suficientes elementos en el multibuffer*
CPRE: Longitud(self) ≥ n
Tomar(self, s, n)
***POST*:** *Retiramos elementos*
POST: n = Longitud(s) ∧ self^{pre} = s + self

PRE: Longitud(s) ≤ $\lfloor MAX/2 \rfloor$
***CPRE*:** *Hay sitio en el buffer para dejar la secuencia*
CPRE: Longitud(self + s) ≤ *MAX*
Poner(self, s)
***POST*:** *Añadimos una secuencia al buffer*
POST: self = self^{pre} + s^{pre}

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-TAD GM

OPERACIONES
ACCIÓN Solicitar: *Tipo_Tamaño[e]* × *Tipo_Dirección[s]*
ACCIÓN Devolver: *Tipo_Tamaño[e]* × *Tipo_Dirección[e]*

SEMÁNTICA
DOMINIO:
TIPO: *GM* = *Tipo_Direccion* ⇸ $\mathbb{B}$
Tipo_Tamaño = 1..Max
Tipo_Dirección = 1..Max
DONDE: *Max* = ...
INICIAL: ∀ $i, 1 \leq i \leq Max \bullet \neg$self($i$)

***CPRE*:** *Hay n posiciones consecutivas de memoria libre*
CPRE: $\exists\, k, 1 \leq k \leq Max - n + 1 \bullet (\forall\, j, k \leq j < k + n \bullet \neg$self($j$))

[7] **Solicitar(n, i)**
POST: *A partir de i había n posiciones de memoria libre que están ahora señaladas como ocupadas*
POST: $\forall\, j \in 0 \ldots n-1 \bullet \neg \text{self}^{pre}(i+j) \wedge \text{self} = \text{self}^{pre} \oplus \{j \mapsto \underline{t} \mid i \leq j < i+n\}$

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

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 × Tipo_Gestor_LE)*

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-TAD Gestor_LE

OPERACIONES
ACCIÓN Iniciar_Lectura:
ACCIÓN Iniciar_Escritura:
ACCIÓN Terminar_Lectura:
ACCIÓN Terminar_Escritura:

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

SEMÁNTICA
DOMINIO:
TIPO: *Gestor_LE = (NLect:* $\mathbb{N}\times$ *Esc:* $\mathbb{B}$*)*
INVARIANTE: $\text{self}.Esc \rightarrow \text{self}.NLect = 0$

INICIAL: $\neg\text{self}.Esc \wedge \text{self}.NLect = 0$

CPRE: $\neg\text{self}.Esc$
Iniciar_Lectura()
POST: $\text{self} = \text{self}^{pre} \setminus \text{self}.NLect = 1 + \text{self}^{pre}.NLect$

[7] También podríamos haber escrito $\text{self} = f \cup \{j \mapsto \underline{f} \mid k \leq j < k+n\}$

CPRE: *cierto*
Terminar_Lectura()
POST: self = self^{pre} \ self.*NLect* = self^{pre}.*NLect* − 1

CPRE: ¬self.*Esc* ∧ self.*NLect* = 0
Iniciar_Escritura()
POST: self = self^{pre} \ self.*Esc*

CPRE: *cierto*
Terminar_Escritura()
POST: self = self^{pre} \ ≠ selfself.*Esc*

Referencias

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

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, FMCO2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer Verlag, 2006.

[CMAHM04] Manuel Carro, Julio Mariño, Ángel Herranz, and Juan José Moreno-Navarro. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In C.N. Dean and R.T. Boute, editors, *Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium*, volume 3294 of *LNCS*, pages 85–106. Springer, 2004. ISBN 3-540-23611-2.

[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.

[HMCM09] Ángel Herranz, Julio Mariño, Manuel Carro, and Juan José Moreno-Navarro. Modeling concurrent systems with shared resources. In *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 102–116, 2009.

[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.

[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.