nueva de bloques será igual a la cola del viejo valor de la cola de bloques, es decir, a todos los elementos de la cola salvo el primero. Una segunda operación es la que se encarga de añadir una cola de bloques *BloquesA* a la cola de bloques. La precondición es que *BloquesA* sea en ese momento un conjunto de bloques usados:

$$BloquesA \subseteq usados$$

¿Cómo se pueden representar las pretonditiones y las posttonditiones?

La postcondición es que el conjunto de bloques se añade al final de la cola de bloques y el conjunto de bloques libres y usados permanece invariable:

$$ColaBloques' = ColaBloques \frown \langle ABlocks \rangle$$
$$usados' = usados$$
$$libres' = libres$$

No cabe duda de que la especificación matemática de la cola de bloques es considerablemente más rigurosa que una narración en lenguaje natural o un modelo gráfico. Este rigor adicional requiere un cierto esfuerzo, pero los beneficios ganados a partir de una mejora de la consistencia y de la completitud puedan justificarse para muchos tipos de aplicaciones.

25.4 LENGUAIES FORMALES DE ESPECIFICACIÓN

Un lenguaje de especificación formal suele estar compuesto de tres componentes primarios: (1) una sintaxis que define la notación específica con la cual se representa la especificación; (2) una semántica que ayuda a definir un «universo de objetos» [WIN90] que se utilizará para describir el sistema; y (3) un conjunto de relaciones que definen las reglas que indican cuáles son los objetos que satisfacen correctamente la especificación.

El *dominio sintáctico* de un lenguaje de especificación formal suele estar basado en una sintaxis derivada de una notación estándar de la teoría de conjuntos y del cálculo de predicados. Por ejemplo, las variables tales como *x*, *y*, y *z* describen un conjunto de objetos que están relacionados a un problema (algunas a veces se denominan el *dominio del discurso)* y se utilizan junto con los operadores descritos en la Sección 25.2. Aunque la sintaxis suele ser simbólica, también se pueden utilizar iconos (símbolos gráficos como cuadros, flechas y círculos) si no son ambiguos.

El *dominio semántico* de un lenguaje de especificación indica la forma en que ese lenguaje representa los requisitos del sistema. Por ejemplo, un lenguaje de programación posee un conjunto de semánticas formales que hace posible que el desarrollador de software especifique algoritmos que transforman una entrada en una salida. Una gramática formal (tal como BNF) se puede utilizar para describir la sintaxis del lenguaje de programación. Sin embargo, un lenguaje de programación no es un buen lenguaje de especificación, porque solamente puede representar funciones computables. Un lenguaje de especificación deberá poseer un dominio semántico más amplio; esto es, el dominio semántico de un lenguaje de especificación debe de ser capaz de expresar ideas tales como «Para todo x de un conjunto infinito *A*, existe un *y* de un conjunto infinito *B* tal que la propiedad *P* es válida para *x e y*» [WIN90]. Otros lenguajes de especificación aplican una semántica que hace posible especificar el comportamiento del sistema. Por ejemplo, se puede desarrollar una sintaxis y una semántica para especificar los estados y las transiciones entre estados, los sucesos y sus efectos en las transiciones de estados, o la sincronización y la temporización.

Es posible utilizar distintas abstracciones semánticas para describir un mismo sistema de diferentes maneras. Eso se ha hecho de manera formal en los Capítulos 12 y 21. El flujo de datos y el procesamiento correspondiente se describía utilizando el diagrama de flujo de datos, y se representaba el comportamiento del sistema mediante un diagrama de transición entre estados. Se empleaba una notación análoga para describir los sistemas orientados a objetos. Es posible utilizar una notación de modelado diferente para representar el mismo sistema. La semántica de cada representación proporciona una visión complementaria del sistema. Para ilustrar este enfoque cuando se utilicen los métodos formales, supóngase que se utiliza un lenguaje de especificación formal para describir el conjunto de sucesos que dan lugar a que se produzca un cierto estado dentro de un sistema. Otra relación formal representa todas aquellas funciones que se producen dentro de un cierto estado. La intersección de estas dos relaciones proporciona una indicación de los sucesos que darán lugar que se produzcan funciones específicas.

En la actualidad se utiliza toda una gama de lenguajes formales de especificación: CSP [HIN95, HOR85], LARCH [GUT93], VDM [JON91] y Z [SPI88, SPI92] son lenguajes formales de especificación representativos que muestran las características indicadas anteriormente. En este capítulo, se utiliza el lenguaje de especificación Z a efectos de ilustración. Z está acompañado de una herramienta automatizada que almacena axiomas, reglas de inferencia y teoremas orientados a la aplicación que dan lugar a pruebas de corrección matemáticas de la especificación.

25.5 USO DEL LENGUAJE Z PARA REPRESENTAR UN COMPONENTE EJEMPLO DE SOFTWARE

Las especificaciones en Z se estructuran como un conjunto de esquemas —son estructuras parecidas a cuadros que presentan variables y que especifican la relación existente entre las variables—. Un esquema es, en esencia, una especificación formal análoga a la subrutina o el procedimiento de un lenguaje de programación. Del mismo modo que los procedimientos y las subrutinas se utilizan para estructurar un sistema, los esquemas se utilizan para estructurar una especificación formal.

En esta sección, se utiliza el lenguaje de especificación Z para modelar el ejemplo del gestor de bloques que se presentaba en la Sección 25.1.3 y se trataba posteriormente en la Sección 25.3. En la Tabla 25.1 se presenta un resumen de la notación del lenguaje **Z.** El siguiente ejemplo de esquema describe el estado del gestor de bloques y del invariante de datos:

La notación Z está basada en la teoría de conjuntos con tipos y en la lógica de primer orden. Z proporciona una estructura denominada ***esquema,*** para describir el estado y las operaciones de una especificación. Los esquemas agrupan las declaraciones de variables con una lista de predicados que limitan los posibles valores de las variables. En ***Z*** el esquema **X** se define en la forma

X

declaraciones

predicados

Las funciones constantes globales se definen en la forma

declaraciones

predecibles

La declaración proporciona el tipo de la función o constante, mientras que el predicado proporciona su valor. En esta tabla solamente se presenta un conjunto abreviado de símbolos de Z.

Conjuntos:	
$S : \mathbb{N}\ X$	S se declara como un conjunto de *X.*
$x \in S$	x es miembro de *S.*
$x \notin S$	x no es miembro de S
$S \subseteq T$	S es un subconjunto de *T*: Todo miembro de S está también en *T.*
$S \cup T$	La unión de S y *T*: Contiene todos los miembros de S o *T o* ambos.
$S \cap T$	La inserción de S y *T*: Contiene todos los miembros tanto de S como de *T.*
$S \setminus T$	La diferencia de S y *T*: Contiene todos los miembros de S salvo los que están también en *T.*
$\emptyset$	Conjunto vacío: No contiene miembros.
{x}	Conjunto unitario: Solamente contiene a x.
$\mathbb{N}$	El conjunto de los números naturales 0, 1, 2....
$S : \mathbb{F}\ X$	Se declara S como un conjunto finito de *X.*
max (*S*)	El máximo del conjunto no vacío de números *S.*
Funciones:	
$f : X \rightarrowtail Y$	Se declara como una inyección parcial de *X* e *Y.*
dom *f*	El dominio de *f*: Dícese del conjunto de valores de x para los cuales está definido *f*(x).
ran *f*	El rango de *f*: El conjunto de valores que toma *f*(x) cuando x recorre el dominio de *f.*
$f \oplus \{x \mapsto y\}$	Una función que coincide con *f* salvo que x se hace corresponder con y.
$\{x\} \lhd f$	Una función igual que *f,* salvo que x se ha eliminado de su dominio.
Lógica:	
P A Q	P y Q: Es verdadero si tanto P como Q son verdaderos.
$P \Rightarrow Q$	P implica Q: Es verdadero tanto si Q es verdadero como si P es falso.
$\theta S' = \theta S$	Ningún componente del esquema *S* cambia en una operación.

TABLA 25.1. Resumen de la notación Z.

Gestionar Bloques

usados, libres: ℙ *BLOQUES*
ColaBloques: seq ℙ *BLOQUES*

---

usados ∩ *libres* = ∅ A
usados ∩ *libres* = *TodosBloques*
∀ *i:* dom *ColaBloques* . *ColaBloques i* ⊆ *usados* A
∀ *i,j :* dom *ColaBloques* · *i* ≠ *j* ⇒
ColaBloques i ∩ *ColaBloques j* = ∅

Referencia Web

Información detallada sobre el lenguaje Z en donde se incluye FAQ se puede encontrar en **archive.comlab. ox.ac.uk/z.html**

El esquema se compone de dos partes. La primera está por encima de la línea central representando las variables del estado, mientras que la que se encuentra por debajo de la línea central describe un invariante de los datos. Cuando el esquema que representa el invariante y el estado se utiliza en otro esquema, va precedido por el símbolo *A*. Por tanto, si se utiliza el esquema anterior en uno que describa, por ejemplo, una operación, se representaría mediante *AGestorBloques.* Como la afirmación anterior establece, los esquemas se pueden utilizar para describir operaciones. El ejemplo siguiente describe la operación que elimina un elemento de una cola de bloques:

Eliminar Bloques

AGestorBloques

#*ColaBloques* > 0,
usados' = *usados* \ *cabeza ColaBloques* A
libres' = *libres* ∪ *cabeza ColaBloques* A
ColaBloques' = *cola ColaBloques*

La inclusión de *AGestorBloques* da como resultado todas las variables que componen el estado disponible en el esquema *EliminarBloques,* y asegura que el invariante de datos se mantendrá antes y después de que se ejecute la operación.

La segunda operación, que añade una colección de bloques al final de la cola, se representa de la manera siguiente:

Añadir Bloques

AGestorBloques
BloquesA? :BLOQUES

BloquesA? ⊆ usados
ColaBloques' = *ColaBloques (BloquesA?)*
usados' = *usados* A
libres' = *libres*

Por convención en Z, toda variable que se lea y que no forme parte del estado irá terminada mediante un signo de interrogación. Consiguientemente, *BloquesA?,* que actúa como parámetro de entrada, acaba con un signo de interrogación.

25.6 MÉTODOS FORMALES BASADOS EN OBJETOS

El interés creciente en la tecnología de objetos ha supuesto que los que trabajan en el área de los métodos formales hayan comenzado a definir notaciones matemáticas que reflejan las construcciones asociadas a la orientación a objetos, llamadas clases, herencia e instanciación. Se han propuesto diferentes variantes de las notaciones existentes, principalmente las que se basan en Z y el propósito de esta sección es examinar Object Z que desarrollaron en el Centro de Verificación de Software de la Universidad de Queensland —.

Object Z es muy similar a Z en detalle. Sin embargo, difiere en lo que se refiere a la estructuración de los esquemas y a la inclusión de las funciones de herencia e instanciación.

A continuación se muestra un ejemplo de especificación en Object Z. Aquí se representa la especificación de una clase que describe una cola genérica que puede tener objetos de cualquier tipo.

Cola[T]

numObjetosMax : ℕ
numObjetosMax 1100

cola: seqT
#*cola* 1*numObjetosMax*

INIT
cola = ⟨⟩

Añadir

Δ(numObjetosMax)
elemento? : T

#cola < numObjetosMax
cola' = cola < elemento? >

Extraer

Δ(numObjetosMax)
elemento! :T

cola ≠ < >
elemento! = cabeza cola
cola' = cola cola

Se ha definido una clase que tiene una variable de instancia *cola,* es decir, una secuencia que tiene objetos del tipo *T,* donde *T* puede ser de cualquier tipo; por ejemplo, puede ser un entero, una factura, un grupo de elementos de configuración o bloques de memoria. Debajo de la definición de cola se leen unos esquemas que definen la clase. La primera define una constante que tendrá un valor no mayor de 100. El siguiente especifica la longitud máxima de la cola y los dos esquemas siguientes *Añadir* y *Extraer* definen los procesos de añadir y extraer un elemento de la cola.

La precondición de *Añadir* especifica que cuando se añade a la cola un objeto *elemento?* no debe tener una longitud mayor a la permitida; y la postcondición especifica lo que ocurre cuando se ha finalizado la adición de elementos. La precondición de la operación *Extraer* especifica que para que esta operación se haga con éxito la cola no debe estar vacía, y la postcondición debe definir la extracción de la cabeza de la cola y su colocación en la variable *elemento!*

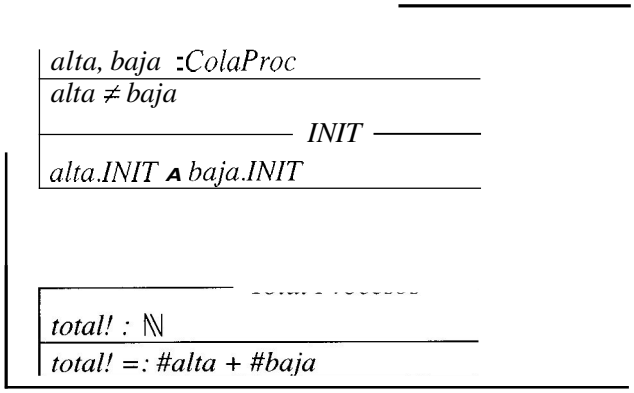
Esto es, por tanto, la especificación básica de la clase de un objeto en Object Z. Si quisiéramos utilizar un objeto definido por dicha clase, sería relativamente simple de hacer. Por ejemplo, supongamos que se está definiendo parte de un sistema operativo que manipulaba procesos que representan los programas en ejecución y que tomamos la decisión de que los procesos tenían que estar en dos colas, una con los procesos de alta prioridad y la otra con aquellos de baja prioridad, y donde la prioridad la utilice el planificador del sistema operativo con el propósito de decidir cual es el siguiente proceso a ejecutar. La primera sentencia de Object Z necesaria instanciará la clase de cola general en una que contenga procesos.

$$ColaProc == Cola[PROCESOS]\ [procQ / cola, proc? / elemento? proc! / elemento!]$$

Se puede decir que todo lo que hace es reemplazar el tipo *T* por *PROCESOS* en la definición de la clase, la cola general *cola* por la cola de los procesos *procQ* y los parámetros generales de entrada y salida *elemento?* y *elemento!,* por aquellos parámetros de proceso *proc?,* y *proc!.* El símbolo / representa una forma de sustitución de texto.

El siguiente paso, como se muestra a continuación, es definir una clase que describe las dos colas. Esta clase utiliza las operaciones definidas en la clase *Cola.* Supongamos que las siguientes operaciones son necesarias:

- *añadirAlta*, añade un proceso a la cola de los procesos de alta prioridad.
- *añadirBaja*, añade un proceso a la cola de los procesos de baja prioridad.
- *INIT* inicializa las colas de alta y baja prioridad.
- *totalProcesos* devuelve el número total de procesos que están en las colas.



La primera línea define el esquema. La segunda y tercera definen el estado y el invariante del estado. En este caso el estado se compone de dos colas de proceso bien diferenciadas: *alta* y *baja.*

A la definición del estado le sigue la definición de cuatro operaciones. *INIT* se define como la aplicación de la operación heredada *INIT* en las colas *alta* y *baja*; *añadirAlta* se define como la aplicación de la operación heredada *añadir* en el componente del estado *alta; añadirBaja* se define como la aplicación de la operación heredada *añadir* sobre el componente de estado *baja;* y, finalmente, la operación *totalProcesos* se define como la suma de tamaños de las colas individuales *alta* y *baja.*

Consecuentemente, esto es un ejemplo de instanciación en acción, donde los objetos definidos por un esquema Object Z se utilizan en otro esquema. Con esto se puede definir la herencia.

Para poder llevar a cabo la definición de herencia, examinemos otro ejemplo, el de una tabla de símbolos. Estas tablas se utilizan constantemente en aplicaciones que contienen los nombres de empleados en sistemas de personal, nombres de impresoras en sistemas operativos, nombres de carreteras en sistemas de navegación para vehículos, y otros. Para describir la herencia, primero definiremos una tabla genérica de símbolos, a continuación la utilizaremos y mostraremos cómo se puede heredar del esquema Z que la define. En primer lugar se deben describir informalmente las operaciones y el estado necesarios.

El estado estará compuesto de un conjunto del tipo T. Suponemos que no habrá más de 100 objetos y definiremos una serie de operaciones que actuarán sobre el estado:

- *INIT* que inicializa la tabla de símbolos para que esté vacía.
- *añadir* que añade un objeto a la tabla de símbolos.
- *extraer* que extrae un objeto de la tabla de símbolos.
- *número* que retorna con el número de objetos de la tabla de símbolos. A continuación se muestra esta definición.

TablaSímbolo[T]

$numObjetosMax : \mathbb{N}$

$MaxElem \leq 100$

$tabla : T$

$\#tabla \leq MaxElem$

INIT

tabla = { }

Añadir

Δ(tabla)
objeto? : T

#tabla < MaxElem
tabla' = tabla ∪ {objeto?}

Extraer

Δ(tabla)
objeto? : T
objeto? ∈ tabla

$tabla' = tabla \setminus (objeto?)$

número

Ξ(tabla)
$numTablaIn! : \mathbb{N}$

$numTablaIn! = \#tabla$

Estas son operaciones muy simples que conllevan operaciones estándar de conjuntos como **u**, y que siguen el formato mostrado en el ejemplo de colas. La operación *añadir* se definirá solo si la tabla tiene espacio para el objeto que se va a añadir, y la operación *extraer* solo se define si el objeto que se va a extraer está dentro de la tabla de símbolos. La operación *número* no afecta al estado, y de aquí que incluya el elemento Ξ(*tabla*).

Si queremos instanciar la tabla como una tabla de nombres de empleados del tipo *EMPLEADO,* se requiere todo lo siguiente:

TabEmpleado == *TablaSimbolos*[*EMPLEADO*]
[*emp/tabla,emp?/objeto?,emp!/objeto!*]

donde la tabla se convierte en una tabla con objetos *EMPLEADO* y donde los parámetros de entrada definidos de forma genérica han sido sustituidos por parámetros específicos *emp?* y *emp!*

A continuación se muestra el esquema Object Z que describe la nueva tabla de empleados mediante la instanciación; simplemente involucra la redefinición de los operadores que se acaban de definir.

tablaEmp

e : TabEmpleados

AñadirEmp ≙ *e.añadir*
ExtraerEmp ≙ *e.extraer*
InitEmp ≙ *e.INIT*
NúmeroEmp ≙ *e.número*

Ahora supongamos que se necesita utilizar la herencia. Para mostrar este caso, supongamos una operación *encontrar* que devuelve un valor *encontrado* si se encuentra un objeto en la tabla de símbolos, y un valor *no encontrado* si no está en la tabla. Vamos a suponer también una aplicación en la que cuando se ejecuta la operación *encontrar* se suele utilizar el mismo objeto que está en la tabla. Aseguraremos que, cuando se aplica antes esta operación, antes de empezar se comprobará que éste es el objeto, lo que podría suponer una búsqueda prolongada en la tabla. Antes de definir este esquema se debe de utilizar el esquema original de tabla de símbolos para incluir esta información. En primer lugar necesitamos un miembro diferenciado de la tabla que se conoce como *FrecElem.* Este es el elemento que se utiliza con más frecuencia. La definición de la tabla nueva es:

TablaDif[T]

TablaSímbolos[T]
FrecElem : T

Encontrar

A(*FrecElem*)
aSerEncontrado? : *T*
estado! : {*encontrado,noencontrado*}

aSerEncontrado? = *FrecElem* ∨ *aSerEncontrado*? ∈ *tabla* ⇒
estado! = *encontrado* ∧ *FrecElem* ' = *aSerEncontrado?*
aSerEncontrado? ≠ *FrecElem* ∧ *aSerEncontrado*? ∉ *tabla* ⇒
estado! = *noEncontrado*

Aquí todas las operaciones definidas para *TablaSímbolos* se heredan sin cambios como está la variable de instancia *tabla.* La nueva operación *encontrar* utiliza una variable *FrecElem* que forma parte del nuevo esquema *TablaDif* de Object Z.

Esta operación comprueba si el parámetro de entrada de *encontrar* es el mismo que el elemento frecuente que se está recuperando varias veces o está dentro de la tabla. **Si** es así, entonces el estado correcto encontrado se devuelve y el elemento frecuente se actualiza. Si el elemento no es el elemento frecuente y no está dentro de la tabla entonces se devuelve el estado *noEncontrado.*

Este esquema se puede utilizar entonces dentro de una aplicación de empleados especificando:

TabEmpleadoFrec == *TablaDif*[*EMPLEADO*]
[*emps/tabla,emp?/objeto?,emp!,Empfrec/Objetofrec*]

Y

EmpleadoFrec

e : TabEmpleadoFrec

AñadirEmpFrec ≙ *e.AÑADIR*
ExtraerEmpFrec ≙ *e.EXTRAER*
InitEmpFrec ≙ *e.INIT*
NúmeroEmpFrec ≙ *e.NUMERO*

Esto es una introducción corta para mostrar la manera de empezar a utilizar las notaciones formales para la especificación de sistemas orientados a objetos. La mayor parte del trabajo que se ha llevado a cabo dentro de esta área ha empleado la noción Z y se ha intentado construir basado en las ideas de estado, precondición, postcondición e invariante de datos, que se ha descrito en la sección anterior, para implementar las funciones para la instanciación y la herencia. El trabajo en esta área está todavía a nivel de investigación con pocas aplicaciones. Sin embargo, durante el período de vida de esta edición las notaciones formales orientadas a objeto deberían experimentar el mismo nivel de penetración industrial que notaciones estándar tales como las notaciones Z.

25.7 ESPECIFICACIÓN ALGEBRAICA

Una de las características principales de las dos técnicas de especificación descritas anteriormente, Z y Z++, es el hecho de que están basadas en la noción de estado: una colección de datos que se ven afectados por operaciones definidas por expresiones escritas en el cálculo del predicado.

Una técnica alternativa se conoce como especificación algebraica. Y consiste en escribir sentencias matemáticas que narran el efecto de las operaciones admisibles en algunos datos. Esta técnica tiene la ventaja principal de apoyar el razonamiento de manera mucho más fácil que los métodos basados en el estado.

El primer paso al escribir una especificación algebraica es determinar las operaciones necesarias. Por ejemplo, podría darse el caso de que un subsistema que implemente una cola de mensajes en un sistema de comunicación necesite operaciones que extraigan un objeto del comienzo de la cola, que devuelvan el número de objetos de una cola y que comprueben si la cola está vacía.

Una vez que se han desarrollado estas operaciones, se puede especificar la relación que existe entre ellas. Por ejemplo, el especificador podría describir el hecho de que cuando se añade un elemento a la cola el número de elemento de esa cola aumenta en un elemento. Para mostrar una impresión del aspecto de una especificación algebraica reproduciremos dos especificaciones. La primera es para una cola, y la segunda para una tabla de símbolos. Asumimos que las operaciones siguientes son necesarias para la cola:

- *ColaVacía.* Devuelve un valor Booleano verdadero si la cola a la que se aplica está vacía y falso si no lo está.
- *añadirObjeto.* Añade un objeto al final de la cola.
- *extraerObjeto.* Extrae un objeto del final de la cola.
- *primero.* Devuelve el primer elemento de una cola, y esta no se ve afectada por esta operación.
- *Último.* Devuelve el último elemento de una cola, y esta no se ve afectada por esta operación.
- *estáVacía?* Devuelve un valor Booleano verdadero si la cola está vacía,' y falso si no lo está.

A continuación se muestran las primeras líneas de la especificación y las operaciones de esta cola:

Nombre: colaParcial
Clases: cola(Z)
Operadores:

colaVacía:	→ cola(Z)
añadirElemento:	Z x cola(Z) → cola(Z)
extraerElemento:	Z x cola(Z) → cola(Z)
primera:	cola(Z) → Z
última:	cola(Z) → Z
estáVacía?:	cola(Z) → Booleana

La primera línea es la que da el nombre al tipo de cola. La segunda línea afirma que la cola manejará cualquier tipo de entrada Z; por ejemplo, Z podría ser mensajes, procesos, empleados esperando entrar a un edificio o aeronaves esperando para aterrizar en un sistema de control del tráfico aéreo.

El resto de la especificación describe las signaturas de las operaciones: cuáles son los nombres y el tipo de elemento que procesan. La definición de *colaVacía* afirma que no toma argumentos, y que da una cola que está vacía; la definición de *añadirElemento* afirma que toma un objeto del tipo Z y una cola con los objetos del tipo Z **y** entonces da una cola de objetos del tipo Z, y así sucesivamente.

Esto es solo la mitad de la especificación—el resto describe la semántica de cada operación en función de la relación con otras operaciones—. **A** continuación, se muestran algunos ejemplos de este tipo de especificación.

La primera afirma que *estáVacía?* será verdadera para una cola vacía y falsa si existe al menos un objeto en la cola; cada una de estas propiedades se expresan en una sola línea

estáVacía?(colaVacía) = verdadera
estáVacía?(añadirObjeto(z,q)) = falsa

La definición de la operación *primera* es

primera(añadirElemento(z,q)) = **z**

la cual establece que *primero* volverá con el primer elemento de la cola.

Con esto se puede ver el estilo de especificación. Para concluir ofreceremos una especificación completa de una tabla de símbolos. Esta es una estructura de datos que contiene una colección de objetos sin duplicados. Estas tablas se encuentran prácticamente en todos los sistemas de computadoras: se utilizan

para almacenar nombres en sistemas de empleados, identificadores de aeronaves en sistemas de control del tráfico aéreo, programas en sistemas operativos y otros.

- *tablaVacía*. Construye una tabla de símbolos vacía.
- *añadirElemento*. Añade un símbolo a una tabla de símbolos que ya exista.
- *extraerElemento*. Extrae un símbolo de una tabla de símbolos que ya exista.
- *estáenTabla?* Será verdadero si un símbolo está en una tabla y falso si no está.
- *unir*. Une el contenido de dos tablas de símbolos.
- *común*. Toma dos tablas de símbolos y retorna con los elementos comunes de cada tabla.
- *esParteDe*. Retorna verdadero si una tabla de símbolos está en otra tabla de símbolos.
- *esIgual*. Retorna verdadero si una tabla de símbolos es igual a otra tabla de símbolos.
- *estáVacía*. Retorna verdadero si la tabla de símbolos está vacía.

A continuación se muestra la definición de las firmas de los operadores

Tabla de nombres

Clases: tablaSímbolos(Z)con =

Operadores:	
tablavacía:	→ tablaSímbolos(Z)
añadirElemento:	Z x tablaSímbolos (Z) → tablaSímbolos (Z)
extraerElemento:	Z x tablaSímbolos (Z) → tablaSímbolos (Z)
estáenTabla?:	Z x tablaSímbolos (Z) → Booleano
unir:	tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)
común:	tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos Z)
esParteDe?:	tablaSímbolos (Z) x tablaSímbolos (Z) → Booleano
esIgual?:	tablaSímbolos (Z) x tablaSímbolos (Z) → Booleano
estáVacía?:	tablaSímbolos (Z) → Booleano

Todas las definiciones se pueden entender al margen de la línea siguiente

Clases: tablaSímbolos (Z) con =

la cual establece que los objetos del tipo Z se asociarán con un operador de igualdad.

La definición del operador *añadirElemento* es

añadirElemento(s2, añadirElemento(s1,s)) =
si s1 = s2 entonces añadirElemento(s2,s)
o si no añadirElemento(s1,añadirElemento(s2,s))

Esto establece que si se realizan dos sumas en la tabla de símbolos y se intenta añadir el mismo elemento dos veces, solo será equivalente a la suma de uno de los dos elementos. Sin embargo, si los elementos difieren, entonces el efecto será el de añadirlos en un orden diferente.

La definición de *extraerElemento* es

extraerElemento(s1,tablaVacía) = tablavacía
extraerElemento (s1, añadirElemento (s2,s)) =
si s1 = s2 entonces extraerElemento (s1,s)
o si no añadirElemento (s2,extraerElemento (s1,s))

Esta definición establece que si se intenta extraer un elemento de una tabla vacía se elaborará la construcción de tabla vacía, y que cuando se extrae un elemento s1 de una tabla que contiene un objeto *s2*, entonces si s1 y *s2* son idénticos el efecto es el de extraer el objeto s1, y si no son iguales el efecto es el de dejar *s2* y extraer el objeto s1.

La definición de *estáenTabla*? es

estáenTabla?(s1, tablavacía) = falso
estáenTabla(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces es verdadero o si no está
en Tabla? (s1,s)

En esta definición se establece que un elemento no puede ser miembro de una tabla de símbolos vacía y que el resultado de aplicar *estaenTabla*? y *s1* a una tabla que esté formada de *s1* y *s2* devolverá un valor verdadero si s1 es igual a *s2*; si no son iguales hay que aplicar *estáenTabla*? a s1.

A continuación se muestra la definición de *unir:*

unir(s, tablavacía) = s
unir(s, añadirElemento(s1,t)) =
añadirElemento(s1,unir (s,t))

Aquí se establece que uniendo una tabla vacía y una tabla *s* da como resultado la construcción de una tabla vacía *s*, y que el efecto de unir dos tablas en donde una de ellas tenga el símbolo s1 es equivalente a añadir s1 al resultado de unir dos tablas.

A continuación se muestra la definición de *común:*

común(s, tablavacía) = tablavacía
común(s,añadirElemento(s1,t)) = si estáenTabla?(s1,s)
entonces añadirElemento(s1,
común(s,t))
o sino común(s,t)

Esta definición establece que la tabla compuesta de los elementos comunes de la tabla vacía y cualquier otra tabla es siempre la tabla vacía. La segunda línea afirma que, si se unen dos tablas, entonces, si hay un elemento común *s1*, este se añade al conjunto común, o de lo contrario se forma el conjunto común de dos conjuntos.

La definición de *esParteDe*? es la siguiente:

esParteDe?(tablaVacía,s) = verdadero
esParteDe?(añadirElemento(s1,s)) =
si estáenTabla? (s1,t) entonces esParteDe?(s,t)
o si no es falso

Esta definición establece que la tabla de símbolos vacía siempre es parte de cualquier tabla de símbolos. La segunda línea establece que si la tabla *t* contiene un elemento en *s* entonces se aplica *esPartede*? para ver si *s* es una subtabla de *t*.

A continuación se presenta la definición de *esIgual*?:

esIgual?(s,t) = esParteDe?(s,t) ^ esPartede?(t,s)

Esta definición establece que las tablas de símbolos son iguales si están dentro una de otra. La definición final de *estáVacía*? es así de sencilla:

estáVacía?(tablaVacía) = verdadero
estáVacía(añadirElemento(s1,t)) = falso

Aquí se establece simplemente que la tabla vacía está vacía y que una tabla que contiene por lo menos un objeto no estará vacía.

Por tanto, la descripción completa de la tabla de símbolos es la siguiente:

Tabla de nombres

Clases: tablaSímbolos(Z) con =

Operadores:

tablavacía: → tablaSímbolos(Z)

añadirTabla: Z x tablaSímbolos (Z) → tablaSímbolos (Z)

extraerElemento: Z x tablaSímbolos (Z) → tablaSímbolos (Z)

estáenTabla?: Z x tablaSímbolos (Z) → Booleana

unir: tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)

común: tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)

esParteDe?: Z x tablaSímbolos (Z) x tablaSímbolos (Z) → Booleana

esIgual?: Z x tablaSímbolos (Z) x tablaSímbolos (Z) → Booleana

estáVacía?: tablaSímbolos (Z) → Booleana

añadirElemento(s2,añadirElemento(s1,s)) =
si s1 = s2 entonces añadirElemento(s2,s)
o si no añadirElemento(s1, añadirElemento(s2,s))

extraerElemento(s1, tablavacía) = tablavacía
extraerElemento(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces extraerElemento(s1,s)
o si no añadirElementos(s2,extraerElemento(s1,s))

estáenTabla?(s1, tablavacía) = falso
estáenTabla?(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces verdadero
o si no estáenTabla?(s1,s)

unir(s, tablavacía) = s
unir(s, añadirElemento(s1,s)) =
añadirElemento(s1, unir(s,t))

común(s, tablavacía) = tablavacía
común(s, añadirElemento(s1,t)) =
si estáenTabla(s1,s)
entonces añadirElemento(s1, común(s,t))
o sino común (s,t)

esParteDe?(tablaVacía,s) = verdadero
esParteDe?(añadirElemento(s1,s),t) =
si estáenTabla?(s1,t) entonces esParte De?(s,t)
o si no falso

esIgual?(s,t) = esParteDe?(s,t) ^ esParteDe?(t,s)

estáVacía?(tablaVacía) = verdadero
estáVacía?(añadirElemento(s1,s) = falso

Tales especificaciones son bastante difíciles de construir, y el problema principal es qúe no se sabe cuándo hay que parar de añadir sentencias que relaten operaciones y que tienden a ser mucho más prolongadas que sus equivalentes basados en el estado. Sin embargo, matemáticamente son más puras y más fáciles para llevar a cabo el razonamiento.

25.8 MÉTODOS FORMALES CONCURRENTES

Las dos secciones anteriores han descrito Z y la derivación orientada a objetos Object Z. El principal problema de estas especificaciones y de las notaciones matemáticas similares es que no tienen las funciones para especificar los sistemas concurrentes: aquellos donde se ejecutan un serie de procesos al mismo tiempo -y- frecuentemente con un grado elevado de comunicación entre estos procesos—. Aunque se han realizado muchos esfuerzos por modificar notaciones tales como Z para acompañar la concurrencia, no se ha hecho con mucho éxito ya que nunca se diseñaron realmente con esta idea en la cabeza. Donde sí se ha hecho con éxito ha sido en el desarrollo de notaciones formales de propósito especial para concurrencia, y el propósito de esta sección es describir una de las más conocidas: PSI (Procesos Secuenciales intercomunicados) [CSP (Communicating Sequential Processes)].

PSI fue desarrollado en Oxford por el científico informático inglés Tony Hoare. La idea principal que respalda esta notación es que se puede ver un sistema como un conjunto de procesos secuenciales (programas simples no concurrentes) que se ejecutan y se comunican con otros procesos de manera autónoma. Hoare diseñó PSI para describir tanto el desarrollo de estas notaciones como la comunicación que tiene lugar entre ellas. De la misma manera que desarrolló esta notación, también desarrolló una serie de leyes algebraicas que permiten llevar a cabo un razonamiento: por ejemplo, sus leyes permiten al diseñador demostrar que el proceso P,no se bloqueará esperando la acción de otro proceso P_2 que está a su vez esperando a que el proceso P, lleve a cabo alguna otra acción.

La notación PSI es muy simple en comparación con una notación Z u Object Z; depende del concepto de un suceso. Un suceso es algo que se puede observar, es atómico e instantáneo, lo que significa que un suceso, por ejemplo, no se puede interrumpir, sino que se completará, independientemente de lo que esté ocurriendo en un sistema. La colección de sucesos asociados con algún proceso *P* se conoce como el alfabeto de *P* y se escribe como αP. Ejemplos típicos de sucesos son el encendido de la válvula de un controlador industrial, la actualización de una variable global de un programa o la transferencia de datos a otra computadora. Los procesos se definen recursivamente en función de los sucesos. Por ejemplo

$$(e \rightarrow P)$$

describe el hecho de que un proceso está involucrado en el suceso *e* dentro de αP y entonces se comporta como *P*. En PSI se pueden anidar definiciones de procesos: por ejemplo, *P* se puede definir en función de otro proceso *P*, como

$$(e \rightarrow (e_a \rightarrow P_1))$$

en donde ocurre el suceso *e,* y el proceso se comporta como el proceso P_1.

La funciones que se acaban de describir no son muy útiles, ya que no proporcionan ninguna opción, por ejemplo, para el hecho de que un proceso se pueda encargar de dos sucesos. El operador de opción | permite que las especificaciones PSI incluyan el elemento de determinismo. Por ejemplo, la siguiente especificación define un proceso *P* que permite una opción.

$$P = (e_1 \rightarrow P_1 | e_2 \rightarrow P_2)$$

Aquí el proceso *P* se define como un proceso capaz de encargarse de dos sucesos e_1 o e_2. Si se da el primero, el proceso se comportará como P_1, y si aparece el segundo, se comportará como P_2.

En PSI existe una serie de procesos estándar. *PARAR* es el proceso que indica que el sistema ha terminado de manera anormal, en un estado no deseado como es el bloqueo. *SALTAR* es un proceso que termina satisfactoriamente. *EJECUTAR* es un proceso que puede encargarse de cualquier suceso en su alfabeto. Al igual que *PARAR* es un proceso no deseado e indica que ha habido un bloqueo.

Para poder hacemos una idea de la utilización de PSI en la especificación de procesos especificaremos algunos sistemas muy simples. El primero es un sistema que se compone de un proceso *C*. Una vez que se ha activado este proceso, la válvula de un reactor químico se cerrará y se parará.

$$\alpha C = \{cerrar\}$$
$$C = (cerrar \rightarrow PARAR)$$

La primera línea define el alfabeto del proceso como un proceso que se compone de un suceso, y la segunda línea establece que el proceso *C* se encargará de *cerrar* y entonces parar.

Esta es una especificación muy simple y algo irreal. Definamos ahora otro proceso C_1 que abrirá la válvula, la cerrará y que comenzará otra vez a comportarseentonces como C_1.

$$\alpha C_1 = \{abrir, cerrar\}$$
$$C_1 = (abrir \rightarrow cerrar \rightarrow C_1)$$

Una definición alternativa donde se definan dos procesos sería

$$\alpha C_1 = \{abrir\}$$
$$C_1 = (abrir \rightarrow C_1)$$

Y

$$\alpha C_2 = (cerrar)$$
$$C_2 = (cerrar \rightarrow C_2)$$

Aquí, el primer proceso C_1 tiene el alfabeto *{abrir),* se encarga de un solo suceso que aparece dentro del alfabeto y que se comporta entonces como el proceso C_2. El C_2 también posee un alfabeto de un solo suceso, se encarga de este suceso y se comporta entonces como C_2. Un observador ajeno al tema que vea el sistema expresado de esta forma vería la siguiente sucesión de sucesos:

abrir, cerrar, abrir, cerrar.. ..

Esta sucesión se conoce como *rastreo* del proceso. A continuación, se muestra otro ejemplo de especificación PSI. Esta representa un robot que se puede encargar de dos sucesos que se corresponden con un retroceso o un avance en la línea. Se puede establecer la definición del camino infinito de un robot sin puntos finales en el recorrido de la siguiente manera

$$\alpha ROBOT = \{avance, retroceso\}$$
$$ROBOT = (avance \rightarrow ROBOT | retroceso \rightarrow ROBOT)$$

Aquí el robot se puede encargar de avanzar o retroceder y comportarse como un robot, pudiendo avanzar y retroceder, y así sucesivamente. Supongamos otra vez que la especificación es real introduciendo algunas otras funciones PSI. La complicación es que la pista sobre la

que viaja el robot se compone de cientos de movimientos con avances y retrocesos que denotan este movimiento. Supongamos también que en esta pista el robot arranca de la posición **1.** A continuación se muestra la definición de este *ROBOT,:*

$$\begin{aligned} \alpha ROBOT_C &= \{\text{avance,retroceso}\} \\ ROBOT, &= R_1 \\ R_1 &= (avance \rightarrow R,) \\ R_i &= (avance \rightarrow R_{i+1} \mid retroceso \rightarrow R,_{i-1}) \\ & \quad (i < 100 \wedge i > 1) \\ R_{100} &= (retroceso \rightarrow R_{99}) \end{aligned}$$

Aquí el robot representa un proceso con los mismos dos sucesos del alfabeto como el robot anterior. Sin embargo, la especificación de **su** implicación en estos sucesos es más compleja. La segunda línea establece que el robot arrancará en una posición inicial y se comportará de la misma manera que con el proceso *R,* el cual representa **su** posición en el primer recuadro. La tercera línea establece que el proceso *R,* solo se puede encargar del suceso de avance ya que se encuentra en el punto final. La cuarta línea define la serie de procesos desde *R,* a R_{99}. Aquí se define el hecho de que en cualquier punto el robot se puede avanzar o retroceder. La línea final especifica lo que sucede cuando el robot ha alcanzado el final del recorrido: solo puede retroceder.

Esta es la manera en que funcionan los procesos en **PSI.** En sistemas reales existirá una serie de procesos ejecutándose concurrentemente y comunicándose con otros, como se muestra a continuación mediante algunos ejemplos:

- En un sistema cliente/servidor, un solo servidor en ejecución como proceso estará en comunicación con varios clientes que igualmente se ejecutarán como procesos.
- En sistemas de tiempo real, que controlan un reactor químico, existirán procesos de supervisión de la temperatura de los reactores que se comunicarán con los procesos que abren y cierran las válvulas de estos reactores.
- En un sistema de control de tráfico aéreo, la funcionalidad del radar se podría implementar como procesos que se comunican con procesos que llevan a cabo las tareas de visualizar en pantalla las posiciones de las aeronaves.

De aquí que exista la necesidad de representar en **PSI** la ejecución en paralelo de los procesos. También existe la necesidad de algún medio con el que se produzca la comunicación y la sincronización entre estos procesos. El operador que especifica la ejecución en paralelo en PSI es ||. Por tanto, el proceso *P* que representa la ejecución en paralelo de los procesos de *P,* y *P,* se define como

$$P = P, \| P,$$

La sincronización entre los procesos se logra mediante procesos con algún solapamiento en sus alfabetos. La norma sobre la comunicación y la sincronización es que cuando dos procesos se están ejecutando en paralelo, donde tienen algunos sucesos en común, deben de ejecutar ese suceso simultáneamente. Tomemos como ejemplo un sistema bastante simple y real que enciende y apaga las luces y que se basa en un ejemplo similar al de [HIN95]. Las definiciones de los dos procesos de este sistema son:

$$\begin{aligned} \alpha LUZ_1 &= \{encendida,apagada\} \\ LUZ, &= (encendida \rightarrow encendida \rightarrow PARAR \\ & \quad \mid apagada \rightarrow apagada \rightarrow PARAR) \\ \alpha LUZ_2 &= \{encendida, apagada\} \\ LUZ, &= (encendida \rightarrow apagada \text{ -+} LUZ,) \end{aligned}$$

Ambos procesos tienen el mismo alfabeto. El primer proceso *LUZ,,* o bien enciende la luz y la vuelve a encender de nuevo (hay que recordar que es posible que otros procesos hayan apagado el sistema entre medias, y se ha averiado *(PARAR* es un estado no deseado), o bien la apaga una vez, y una vez más. El proceso *LUZ,* enciende la luz y la apaga, y a continuación empieza a comportarse de nuevo como *LUZ,.* Los dos procesos en paralelo se denotan mediante:

$$LUZ, \| LUZ,$$

¿Cuál es el efecto de ejecutar estos procesos en paralelo? En una introducción como es esta no se puede entrar en mucho detalle. Sin embargo, se puede dar una idea del razonamiento que se puede aplicar a dicha expresión. Se recordará que cuando se introdujo PSI se afirmó que no solo consiste en una notación para especificar los procesos concurrentes, sino que también contiene una serie de normas que permiten razonar a cerca de las expresiones de procesos complejos y, por ejemplo, determinar si cualquier suceso nefasto como un bloqueo aparecerá dentro del sistema especificado en PSI. Para poder ver lo que ocurre merece la pena aplicar algunas de las normas que desarrolló Hoare para PSI. Recordemos que se intenta averiguar cuál es el proceso que se ha definido mediante la ejecución en paralelo de los procesos *LUZ,* y *LUZ,.* Esta expresión es equivalente a:

$$\begin{aligned} LUZ_1 &= (encendido \rightarrow encendido \text{ -+} PARAR \\ & \quad \mid apagado \rightarrow apagado \rightarrow PARAR) \\ LUZ, &= (encendido\text{-+} apagado \rightarrow LUZ_2) \end{aligned}$$

Una de las normas de Hoare establece que

$$(e \rightarrow P) \| (e \rightarrow Q) = e \rightarrow (P \| Q)$$

.Esto nos permite ampliar la expresión que involucra a LUZ, y LUZ, con

$$(encendido \rightarrow ((encendido \rightarrow PARAR) \| (apagado \rightarrow LUZ_2)))$$

Esta expresión se puede simplificar aun más utilizando otra de las normas de Hoare a la expresión

$$(encendido \text{ -+} PARAR)$$

lo cual significa que la ejecución en paralelo de los dos procesos es equivalente a una luz que se enciende **y** entonces se para en un estado de bloqueo no deseado.

Hasta ahora, se han visto procesos que cooperan con la sincronización mediante el hecho de que tienen alfabetos similares o que se solapan. En los sistemas reales también existirá la necesidad de que los procesos se comuniquen con datos. **PSI** es un método uniforme para la comunicación de datos: se trata simplemente como un suceso en donde el suceso *nomCanal.val* indica que ha ocurrido un suceso que se corresponde con la comunicación del valor *val* a través de un canal *nomCanal*. PSI contiene un dispositivo notacional similar al de Z para distinguir entre la entrada y la salida; para distinguir la primera se introduce el signo de interrogación, mientras que para distinguir la segunda se utiliza el signo de exclamación.

A continuación, se muestra un ejemplo basado en [HIN95], en donde se representa la especificación de un proceso que toma dos enteros y forma el producto.

$$
\begin{aligned}
PROD, &= A_{\langle\rangle} \\
&= (en?\,x \rightarrow \mathbf{A}_{,,,}) \\
A_{\langle x\rangle} &= (en?\,y \rightarrow A_{(x,y)}) \\
A_{\langle x,y\rangle} &= (fuera!(x * y) \rightarrow \mathbf{A}_{,,,})
\end{aligned}
$$

Aprimera vista esto parece muy complicado, por eso merece la pena describirlo línea a línea. La primera define el proceso *PROD,,* el cual equipara este proceso con **A,** sin entradas esperando. La segunda línea establece que cuando el proceso recibe un valor de entrada x se comporta como el proceso con un valor x almacenado. La tercera línea establece que cuando un proceso con un valor almacenado recibe un valor y entonces se comporta como un proceso con dos valores almacenados. La última línea establece que un proceso con dos valores almacenados emitirá el producto de estos valores, almacenará el último valor y entonces se comportará como **A** con ese valor almacenado, de manera que, por ejemplo, si aparece otro valor se multiplicará por ese valor y se emitirá por el canal de salida. **Así** pues, la especificación anterior se comporta como un proceso en donde se recibe una sucesión de enteros y lleva a cabo la multiplicación de los valores.

Se puede decir entonces que esta es una definición breve de PSI —al igual que todos los métodos formales incluye también una notación matemática y las normas para el razonamiento—. Aunque en las descripciones de Z y Object Z no se han examinado las normas y se ha concentrado en el formalismo todavía contienen funciones sustanciales para razonar sobre las propiedades de un sistema.

25.9 LOS DIEZ MANDAMIENTOS DE LOS MÉTODOS FORMALES

La decisión de hacer uso de los métodos formales en el mundo real no debe de adoptarse a la ligera. Bowen y Hinchley [BOW95] han acuñado «los diez mandamientos de los métodos formales», como guía para aquellos que estén a punto de embarcarse en este importante enfoque de la ingeniería del software[5].

La decisión de utilizar métodos formales no debería tomarse a lo ligera. Siga estos «mandamientos» y asegúrese de que todo el mundo haya recibida lo formación adecuada.

1. **Seleccionarás la notación adecuada.** Con objeto de seleccionar eficientemente dentro de la amplia gama de lenguajes de especificación formal existente, el ingeniero del software deberá considerar el vocabulario del lenguaje, el tipo de aplicación que haya que especificar y el grado de utilización del lenguaje.
11. **Formalizarás, pero no de más.** En general, resulta necesario aplicar los métodos formales a todos los aspectos de los sistemas de cierta envergadura. Aquellos componentes que sean críticos para la seguridad serán nuestras primeras opciones, e irán seguidos por aquellos componentes cuyo fallo no se pueda admitir (por razones de negocios).
111. **Estimarás los costes.** Los métodos formales tienen unos costes de arranque considerables. El entrenamiento del personal, la adquisición de herramientas de apoyo y la utilización de asesores bajo contrato dan lugar a unos costes elevados en la primera ocasión. Estos costes deben tenerse en cuenta cuando se esté considerando el beneficio obtenido frente a esa inversión asociada a los métodos formales.
IV. **Poseerás un experto en métodos formales a tu disposición.** El entrenamiento de expertos y la asesoría continua son esenciales para el éxito cuando se utilizan los métodos formales por primera vez.
V. **No abandonarás tus métodos formales de desarrollo.** Es posible, y en muchos casos resulta deseable, integrar los métodos formales con los métodos convencionales y/o con métodos orientados a objetos (Capítulos 12y 21). Cada uno de estos métodos posee **sus** ventajas y **sus** inconvenientes. Una combinación de ambos, aplicada de forma adecuada, puede producir excelentes resultados[6].

[5] Esta descripción es una versión sumamente abreviada de [BOW95].

[6] La ingeniería del software de la sala limpia (Capítulo *26)* es un ejemplo integrado que hace uso de los métodos formales y **de** una notación más convencional para el desarrollo..

Referencia Web

Información útil sobre los métodos formales se puede obtener en: **www.dcam/users/mgh1001**

VI. Documentarás suficientemente. Los métodos formales proporcionan un método conciso, sin ambigüedades y consistente para documentar los requisitos del sistema. Sin embargo, se recomienda que se adjunte un comentario en lenguaje natural a la especificación formal, para que sirva como mecanismo para reforzar la comprensión del sistema por parte de los lectores.

VII. No comprometerás los estándares de calidad. «Los métodos formales **no** tienen nada de mágico» [BOW94], y, por esta razón, las demás actividades de **SQA** (Capítulo 8) deben de seguir aplicándose cuando se desarrollen sistemas.

VIII. No serás dogmático. El ingeniero de software debe reconocer que los métodos formales no son una garantía de corrección. Es posible (o como algunos probablemente dirían) que el sistema final, aun cuando se haya desarrollado empleando métodos formales, siga conteniendo pequeñas omisiones, errores de menor importancia y otros atributos que no satisfagan nuestras expectativas.

IX. Comprobarás, comprobarás y volverás a comprobar. La importancia de la comprobación del software se ha descrito en los Capítulos 17, 18 y 23. Los métodos formales no absuelven al ingeniero del software de la necesidad de llevar a cabo unas comprobaciones exhaustivas y bien planeadas.

X. Reutilizarás cuanto puedas. A la larga, la única forma racional de reducir los costes del software y de incrementar la calidad del software pasa por la reutilización (Capítulo 27). Los métodos formales no modifican esta realidad. De hecho, **quizás** suceda que los métodos formales sean un enfoque adecuado cuando es preciso crear componentes para bibliotecas reutilizables.

25.10 MÉTODOS FORMALES: EL FUTURO

Aun cuando las técnicas de especificación formal, con fundamento matemático, todavía no se utilizan con demasiada frecuencia en la industria) éstas ofrecen ciertamente unas ventajas substanciales con respecto a las técnicas menos formales. Liskov y Bersins [LIS86] resumen esto en la manera siguiente:

> Las especificaciones formales se pueden estudiar matemáticamente, mientras que las especificaciones informales no pueden estudiarse de esta manera. Por ejemplo, se puede demostrar que un programa correcto satisface sus especificaciones, o bien se puede demostrar que dos conjuntos alternativos de especificaciones **son** equivalentes... Ciertas formas de falta de completitud o de inconsistencia se pueden detectar de forma automática.

Además, la especificación formal elimina la ambigüedad, y propugna un mayor rigor en las primeras fases del proceso de ingeniería del software.

Pero siguen existiendo problemas. La especificación formal se centra fundamentalmente en las funciones y los datos. La temporización, el control y los aspectos de comportamiento del problema son más difíciles de representar. Además, existen ciertas partes del problema (por ejemplo, las interfaces hombre-máquina) que se especifican mejor empleando técnicas gráficas. Por último, la especificación mediante métodos formales es más difícil de aprender que otros métodos de análisis que se presentan en este libro y representa «un choque cultural)) significativo para algunos especialistas del software. Por esta razón, es probable que las técnicas de especificación formales matemáticas pasen a ser el fundamento de una futura generación de herramientas CASE. Cuando esto suceda, es posible que las especificaciones basadas en matemáticas sean adoptadas por un segmento más amplio de la comunidad de la ingeniería del software[7].

RESUMEN

Los métodos formales ofrecen un fundamento para entornos de especificación que dan lugar a modelos de análisis más completos, consistentes y carentes de ambigüedad, que aquellos que se producen empleando métodos convencionales u orientados a objetos. Las capacidades descriptivas de la teoría de conjuntos y de la notación lógica capacitan al ingeniero del software para crear un enunciado claro de los hechos (requisitos).

Los conceptos subyacentes que gobiernan los métodos formales son: (1) los invariantes de datos - c o n -

[7] Es importante tener en cuenta que hay otras personas que no están de acuerdo. Véase [YOU94].

diciones que son ciertas a lo largo de la ejecución del sistema que contiene una colección de datos —; (2) el estado —los datos almacenados a los que accede el sistema y que son alterados por él—; (3) la operación —una acción que tiene lugar en un sistema y que lee o escribe datos en un estado—. Una operación queda asociada con dos condiciones: una precondición y una postcondición.

La matemática discreta —la notación y práctica asociada a los conjuntos y a la especificación constructiva, a los operadores de conjuntos, a los operadores lógicos y a las sucesiones —constituyen la base de los métodos formales. Estas matemáticas están implementadas en el contexto de un lenguaje de especificación formal, tal como Z.

Z, al igual que todos los lenguajes de especificación formal, posee tanto un dominio semántico como un dominio sintáctico. El dominio sintáctico utiliza una simbología que sigue estrechamente a la notación de conjuntos y al cálculo de predicados. El dominio semántico capacita al lenguaje para expresar requisitos de forma concisa. La estructura Z contiene esquemas, estructuras en forma de cuadro que presentan las variables y que especifican las relaciones entre estas variables.

La decisión de utilizar métodos formales debe considerar los costes de arranque, así como los cambios puntuales asociados a una tecnología radicalmente distinta. En la mayoría de los casos, los métodos formales ofrecen los mayores beneficios para los sistemas de seguridad y para los sistemas críticos para los negocios.

REFERENCIAS

[BOW95] Bowan, J.P., y M.G. Hinchley, «Ten Commandments of Formal Methods, *Computer*», vol. 28, n.º 4, Abril 1995.

[GRI93] Gries, D., y F.B. Schneider, *A Logical Approach to Discrete Math,* Springer-Verlag, 1993.

[GUT93] Guttag, J.V., y J.J. Horning, *Larch: Languages and Toolsfor Formal Specifications*, Springer-Verlag, 1993.

[HAL90] Hall, A., «Seven Myths of Formal Methods», IEEE Software, Septiembre 1990.

[HOR85] Hoare, C.A.R., *Communicating Sequential Processes,* Prentice-Hall International, 1985.

[HIN95] Hinchley, M.G., y S.A. Jarvis, *Concurrent Systems: Formal Development in CSP,* McGraw-Hill, 1995.

[JON91] Jones, C.B., *Systematic Development Using VDM,* 2.ª ed., Prentice-Hall, 1991.

[LIS86] Liskov, B.H., y V. Berzins, «An Appraisal of Program Specifications», publicado en *Software Specifications Techniques;* eds.: N. Gehani y A.T. McKittrick, Addison-Wesley, 1986, p. 3.

[MAR94] Marcianiak, J.J. (ed.), *Encyclopedia of Software Engineering,* Wiley, 1994.

[ROS95] Rosen, K.H., *Discrete Mathernatics and Its Applications,* 3.ª ed., McGraw-Hill, 1995.

[SPI88] Spievy, J.M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, 1988.

[SPI92] Spievy, J.M., *The Z Notation: A Reference Manual,* Prentice-Hall, 1992.

[WIL87] Witala, S.A., *Discrete Mathematics: A Unified Approach,* McGraw-Hill, 1987.

[WIN90] Wing, J.M., «A Specifier's Introduction to Formal Methods», *IEEE Computer*, vol. 23, n.º 9, Septiembre 1990, pp. 8-24.

[YOU94] Yourdon, E., «Formal Methods», *Guerrilla Programmer*, Cutter Information Corp., Octubre 1994.

PROBLEMAS Y PUNTOS A CONSIDERAR

25.1. Revisar los tipos de deficiencias asociados a los enfoques menos formales de la ingeniería del software en la Sección 25.1.1. Proporcione tres ejemplos de cada uno de ellos, procedentes de su propia experiencia.

25.2. Los beneficios de las matemáticas como mecanismo de especificación se han descrito con cierta extensión en este capítulo. ¿Existe algún aspecto negativo?

25.3. Se le ha asignado un equipo de software que va a desarrollar software para un fax módem. Su trabajo consiste en desarrollar el «listín telefónico» de la aplicación. La función del listín telefónico admite hasta *MaxNombre* nombres de direcciones que serán almacenados junto con los nombres de la compañía, números de fax y otras informaciones relacionadas. Empleando el lenguaje natural, defina:

a. el invariante de datos

b. el estado

c. las operaciones probables

25.4. Se le ha asignado un equipo de software que está desarrollando software, denominado *DuplicadosMemoria*, y que proporciona una mayor cantidad de memoria aparente para un PC, en comparación con la memoria física. Esto se logra identificando, recogiendo y reasignando bloques de memoria que hayan sido asignados a aplicaciones existentes pero no estén siendo utilizados. Los bloques no utilizados se reasignan a aplicaciones que requieran memoria adicional. Efectuando las suposiciones oportunas, y empleando el lenguaje natural, defina:

a. el invariante de datos

b. el estado

c. las operaciones probables

25.5. Desarrollar una especificación constructiva para un conjunto que contenga tuplas de números naturales de la forma (x, y, z^2) tales que la suma de x e y es igual a z.

25.6. El instalador de una aplicación basada en PC determina si está presente o no un conjunto de recursos de hardware y software adecuados. Comprueba la configuración de hardware para determinar **si** están presentes o no diferentes dispositivos (de entre muchos dispositivos posibles) y determina si ya están instaladas las versiones específicas de software del sistema y de controladores. ¿Qué conjunto de operadores se utilizaría para lograr esto? Proporcionar un ejemplo en este contexto.

25.7. Intente desarrollar una expresión empleando la lógica y un conjunto de operadores para la siguiente sentencia: «Para todo x e y, si x es padre de y e y es padre de z, entonces x es abuelo de z. Todas las personas tienen un padre.» Pista: utilice las funciones $P(x, y)$ y $G(x, z)$ para representar las funciones padre y abuelo, respectivamente.

25.8. Desarrollar una especificación constructiva de conjuntos correspondiente al conjunto de parejas en las cuales el primer elemento de cada pareja es la suma de dos números naturales no nulos, y el segundo elemento es la diferencia de esos dos números. Ambos números deben de estar entre 100 y 200 inclusive.

25.9. Desarrollar una descripción matemática del estado y del invariante de datos para el Problema 25.3. Refinar esta descripción en el lenguaje de especificación **Z.**

25.10. Desarrollar una descripción matemática del estado y del invariante de datos para el Problema 25.4. Refinar esta descripción en el lenguaje de especificación **Z.**

25.11. Utilizando la notación **Z** presentada en la Tabla 25.1, seleccionar alguna parte del sistema de seguridad *HogarSeguro* descrito anteriormente en este libro, e intentar especificarlo empleando **Z.**

25.12. Empleando una o más de las fuentes de información indicadas en las referencias de este capítulo o en la sección de *Otras Lecturas y Fuentes de Información,* desarrollar una presentación de media hora acerca de la sintaxis y semántica básica de un lenguaje de especificación formal y distinto de **Z.**

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Además de los libros utilizados como referencia en este capítulo, se ha publicado un número bastante grande de libros acerca de temas relacionados con **los** métodos formales a lo largo de los últimos años. Se presenta a continuación un listado de los ejemplos más significativos:

Bowan, J., *Formal Specification and Documentation using Z: A Case study Approach,* International Thomson Computer Press, 1996.

Casey, C., *A Programming Approach to Formal Methods,* McGraw-Hill, 2000.

Cooper, D., y R. Barden, *Z in Practice,* Prentice-Hall, 1995.

Craigen, D., S. Gerhart y T. Raltson, *Industrial Application of Formal Methods to Model, Design, Diagnose and Analize Computer Systems,* Noyes Data Corp., Park Ridge, NJ, 1995.

Diller, A., *Z.: An Introduction to Formal Methods,* 2.ª ed., Wiley, 1994.

Harry, A., *Formal Methods Fact File: VDM y Z,* Wiley, 1997

Hinchley, M., y J. Bowan, *Applications of Formal Methods,* Prentice-Hall, 1995.

Hinchley, M., y J. Bowan, *Industrial Strength Formal Methods,* Academic Press, 1997.

Hussmann, H., *Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997.

Jacky, J., The Way of Z: *Practical Programming With Formal Methods,* Cambridge University Press, 1997.

Lano, J., y Haughton (eds.), *Object-Oriented Specification Case Studies,* Prentice-Hall, 1993.

Rann, D., J. Turner y J. Whitworth, *Z: A Beginner's Guide,* Chapman & Hall, 1994.

Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach,* McGraw-Hill, 1994.

D. Sheppard, *An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995.

Los ejemplos de septiembre de 1990 de *IEEE Transactions on Software Engineering, IEEE Software e IEEE Computer* estaban dedicados todos ellos a los métodos formales. Siguen siendo una fuente excelente de información Útil.

Schuman ha editado un libro que describe los métodos formales y las tecnologías orientadas a objetos *(Formal Object-Oriented Development,* Springer-Verlag, 1996). El libro ofrece líneas generales acerca de la utilización selectiva de métodos formales y acerca de la forma en que estos métodos se pueden utilizar en conjunción con enfoques OO.

En Internet se encuentra una gran cantidad de información acerca de los métodos formales y de otros temas relacionados. En **http://www.pressman5.com** se puede encontrar una lista de referencias actualizada relevante para los métodos formales.