PROGRAMACIÓN METÓDICA

1.− ESPECIFICACIÓN. INSTRUCCIONES BÁSICAS.

DERIVACIÓN.

2.− VERIFICACIÓN Y DERIVACIÓN.

3.− PRINCIPIO DE INDUCCIÓN

4.− PROGRAMAS RECURSIVOS.

5.− INMERSIÓN DE ALGORISMOS

6.− PROFUNDIZACIÓN EN PROGRAMAS ITERATIVOS.

7.− VARIOS, APLICACIONES

1.− ESPECIFICACIÓN. INSTRUCCIONES BÁSICAS.

Para comprobar si un programa ha sido escrito adecuadamente se suelen efectuar algunas ejecuciones de prueba que permitan conocer su comportamiento. Intentamos saber si calcula los resultados deseados, y es, por tanto, correcto.

Mediante estas ejecuciones, en efecto, se puede detectar la presencia de algún error, y en muchos casos resulta sorprendentemente laboriosa su identificación y reparación, sobre todo en caso de programas extensos y complejos. Pero puede darse un caso aún peor: que las pruebas no revelen errores, y éstos aparezcan más tarde, cuando el daño es mayor.

Un segundo inconveniente que ofrece esta validación dinámica es que, una vez detectada la existencia de errores, la información que obtenemos sobre los puntos del programa en que se encuentran es muy escasa.

La alternativa será una validación estática, que consista en obtener información a priori sobre el comportamiento del programa mediante el análisis de su propio texto. Este análisis debe ser capaz de demostrar que los resultados que el programa proporciona son los deseados, o bien de detectar la presencia de errores e identificarlos completamente; este proceso se llama **verificación**.

La construcción de programas ya verificados se denomina derivación, y resulta mucho más importante en la práctica que la verificación.

La corrección se define entonces como la coincidencia entre el comportamiento deseado del programa y su comportamiento real.

Para expresar el comportamiento esperado de un algoritmo usaremos especificaciones. Estas constarán de una precondición (condiciones que deben cumplir los datos del programa) y una postcondición (relaciones que deben existir entre los datos recibidos y los resultados obtenidos).

Un algoritmo es considerado como una sucesión de estados con instrucciones entre cada estado.

ESTADOS, ASERTOS Y EXPRESIONES

El análisis de un programa puede hacerse en términos de un conjunto de estados iniciales admisibles y un conjunto de estados finales apropiados. Para describir conjunto de estados utilizaremos asertos.

Dado un conjunto de variables (que pueden o no tener un valor), un **estado** es una aplicación de este conjunto de variables en el conjunto de sus valores, de manera que se asocie a cada variable un valor coherente con su tipo. Si se observa el valor de las variables de un programa en un cierto momento obtenemos el estado del programa en ese momento.

Un **aserto**, **predicado** o **aserción** es una expresión lógica que involucra las variables del programa que usamos para expresar el comportamiento de dicho programa en ciertos momentos. Los asertos no son las únicas expresiones lógicas que usamos. Hay asertos no evaluables (no pueden aparecer en las instrucciones del programa), estas expresiones son las expresiones cuantificadas (expresiones introducidas por un cuantificador) y las operaciones ocultas (expresiones expresadas con un nombre).

CUANTIFICDORES Y VARIABLES

Un **cuantificador** es una abreviatura de una secuencia de operaciones que requiere ir asociado a una **variable ligada o ciega**, que es simplemente un identificador, y a un **dominio**, que indica el conjunto de valores que permitimos tomar a la variable ligada y que frecuentemente es un intervalo de los naturales. La expresión así obtenida denota la repetición de la operación sobre todos los valores del dominio. Usaremos con frecuencia los siguientes cuantificadores:

−" : dominio: E( )

" : dominio: E( )

denotan, respectivamente, la suma y el producto de todos los valores tomados por la expresión E, cuando recorre todos los valores indicados en el dominio.

−" : dominio: E( )

" : dominio: E( )

denotan, la conjunción y la disyunción de todos los valores que toma la expresión E, cuando recorre todos los valores indicados en el dominio.

$-N$: dominio: E( )

denota el número de valores en el dominio de para los que E es cierta.

− MAX : dominio: E( )

MIN : dominio: E( )

denotan el mayor y el menor de todos los valores que toma la expresión E( ), cuando recorre el dominio. La variable ligada en estos casos seria la letra griega ; puede usarse cualquier otra variable en su lugar sin que cambie el significado de la expresión.

Es concebible aplicar algunos cuantificadores sobre un dominio vacío. El resultado del contador sobre un dominio vacío es cero. La iniciación de un bucle de la iteración y el caso directo de la recursividad se realizan mediante el neutro o el dominio vacío de los cuantificadores.

Cuantificadores	**Símbolo**	**Estructura**	**Tipo**	**Neutro**	**Comentarios.**
Sumatorio	"	" : 1< <n: a( )	numero	0	
Productorio	"	" : 1< <n: a( )	numero	1	
Universal	"	" : 1< <n: a( )=0	booleano	cierto	Abrevia la conjunción
Existe	"	" : 1< <n: a( )=k	booleano	falso	Abrevia la disyunción
Contador	*N*	*N* : 1< <n: a( )	natural	0	
Máximo	MAX	MAX : 1< <n: a( )	numero	X	
Mínimo	MIN	MIN : 1< <n: a( )	numero	X	

Cuadro resumen de los cuantificadores

Cuando el dominio no es nulo, podemos separar uno de sus elementos y reducir en uno el dominio del cuantificador. El elemento separado se combina con el cuantificador reducido, mediante la correspondiente operación binaria.

(" : 1" "n: a( )) = (" : 1" "n−1: a( )+a(n))

(" : 1" "i: a( )) + (" : i< "n: a( ))

(MAX : 1" "n: a( )) = max (MAX : 1" "n−1: a( ), a(n))

De la misma manera, podemos separar un elemento arbitrario del dominio de un cuantificador cualquiera, salvo que sea vacío, combinándolo con el resto mediante la operación binaria asociada al cuantificador.

El cuantificador contador para poder separarlo en dos se necesita una alternativa, una condición.

Las variables que aparezcan en los asertos podrán ser de los siguientes tipos:

−*Variables ligadas o ciegas*: está vinculada a un cuantificador, y puede ser sustituida por cualquier otra sin que se modifique el significado de la expresión.

−*Variables libres*: dentro de este grupo se encuentran:

·Del programa: que denotan, en cada punto del mismo, el último valor que se les haya asignado.

·Iniciales: se usan para denotar un valor que sólo se conoce en un punto diferente de la ejecución del programa.

Conviene evitar el uso del mismo nombre para variables del programa y variables ligadas.

Cada aserto puede ser considerado como una descripción de un conjunto de estados: aquellos en los que los valores de las variables hacen que el aserto evalúe a cierto.

SUSTITUCIONES

Una importante operación que se puede realizar sobre los asertos es la de sustitución, que permitirá razonar sobre instrucciones de asignación.

En ningún momento se debe utilizar una variable ligada o ciega y una libre con el mismo nombre, ni siquiera en distintos asertos o expresiones. De no ser así basta dar nuevos nombres a las variables ligadas manteniendo el significado.

Sea A un aserto, x una variable y E una expresión del mismo tipo que x. Denotaremos:

AxE

el aserto que se obtiene sustituyendo todas las menciones de la variable x en A por la expresión E.

Al hacer un cambio se ha de ir con cuidado de no poner los mismos símbolos a las variables ligadas y a las variables libres ya que podrían cambiar el significado el aserto.

ESPECIFICACIÓN PRE/POST

Al expresar, mediante un aserto, las condiciones que se imponen a los datos, se está restringiendo el conjunto de estados en que se puede poner en marcha el programa con garantías de funcionamiento correcto. Este aserto que impone condiciones a los datos s denomina **precondición**. Cuanto más débil sea esta precondición, más útil será el programa, ya que podrá emplearse en más casos. De manera dual, el aserto que expresa las propiedades de los resultados del algoritmo se denomina **postcondición**.

Una especificación pre/post para un programa P se escribe: {A} P {B}

y denota que se requiere del programa P que, si comienza a funcionar en un estado que satisfaga A, termine en un tiempo finito y en un estado que satisfaga B. La especificación describe el comportamiento esperado del programa P.

Verificar el programa consiste entonces en demostrar que cumple su especificación.

Supongamos que se ha demostrado {A} P {B}. Entonces:

−Si A!A' entonces {A'} P {B} es cierto y además se refuerza el aserto.

−Si B!B' entonces {A} P {B'} es cierto y además se debilita el aserto.

Frecuentemente el fragmento de programa a diseñar no va a considerarse aislado, sino que es parte del desarrollo de un programa más complejo. En este caso, puede ser conveniente dar un nombre a este fragmento de programa, y seleccionar unos parámetros de entrada y unos resultados. Lo haremos mediante la **función**.

Para especificar la función necesitamos saber qué variables representan datos y cuales resultados. La especificación constará de tres partes: cabecera, pre y postcondición. La cabecera indica el nombre de la función y como se llaman y de que tipo son los *parámetros* y las variables locales en las que se calcularán los *resultados*. La precondición involucrará los parámetros y la postcondición los resultados.

La sintaxis para presentar la cabecera de una función es:

funció nom (parámetros) *ret* resultados

{Pre}

variables locales

{Post}

ret resultados

ffunció

Parámetros o resultados d distintos se separan por ;.

Una vez calculados los resultados, todos ellos aparecerán al final del cuerpo de la función en una instrucción dev o ret, que representa la devolución de resultados al punto en que se produjo la llamada. Sólo puede haber una instrucción dev o ret en cada función, y ha de estar situada como última instrucción a realizar.

TEMA 2: VERIFICACIÓN Y DERIVACIÓN

SEMÁNTICA AXIOMÁTICA

En esta sección se presenta la definición del lenguaje de programación:

−*Instrucción nula*: corresponde a continuar la ejecución sin hacer nada, es decir, sin modificar el estado de las variables. Su denotación es:

continuar, seguir o "

y su semántica viene definida por {A} " {A}. Es decir todo lo que se cumple antes se sigue cumpliendo después.

−*Asignación simple*: modificación de una determinada variable, y por lo tanto conlleva una modificación del estado. La asignación simple sólo afecta a una variable. La instrucción se denota escribiendo la variable que recibe el valor a la izquierda del símbolo := y la expresión que denota el valor a la derecha: {A} x:=*E* {B}

La variable x ha de ser del mismo tipo que la expresión E.

Para demostrar que la asignación es correcta hay que demostrar que E puede evaluarse sin errores y que A ! BxE que expresa en A ha de implicar B cambiando las variables x por E.

−*Asignación múltiple*: Dada una cierta cantidad de variables distintas y expresiones y suponiendo que los tipos coinciden denotamos:

<x1, x2,....xn> := <E1, E2,....En>

la instrucción de asignación múltiple, simultánea, del valor de cada expresión a su variable correspondiente. La característica de simultaneidad es importante, dado que es factible que algunas de las variables x aparezcan en algunas de las expresiones E y deben ser interpretadas con el valor que tengan previamente a la asignación.

−*Composición*: es la composición secuencial de otras acciones para ser ejecutadas en el orden indicado. Se denota separando las acciones que la forman con un punto y coma: P1;P2. Su definición semántica es como sigue: para demostrar {A1} P1;P2 {A3} es preciso encontrar un A2 tal que se pueda demostrar {A1} P1 {A2} y {A2} P2 {A3}.

−*Alternativa*: esta instrucción permite seleccionar acciones a ejecutar en función del estado en que se encuentre el algoritmo. Se basa en el concepto de **instrucción protegida**, que es un par formado por una expresión booleana llamada protección y una instrucción cualquiera; si la protección evalúa a cierto diremos

que está **abierta** y **cerrada** en caso contrario. La idea es que sólo estará permitido ejecutar una instrucción protegida si su protección está abierta. La sintaxis será:

{Pre=A1}

si

B1 −−> S1

.

.

Bn −−> Sk

fsi

{Post=A2}

donde cada rama corresponde a una instrucción protegida; por tanto las expresiones Bi habrán de ser tipo booleano. Se ejecuta evaluando todas las protecciones; si todas ellas están cerradas, la ejecución del algoritmo aborta; y en caso contrario se selecciona indeterminístacmente una protección abierta y se ejecuta su instrucción asociada.

Para comprobar si la precondición A1 es válida respecto de la postcondición A2, han de verificarse los siguientes puntos:

−Que al menos una de las protecciones esté abierta: A1 ! B1"B2"..."Bn

−Que cualquier protección abierta dé lugar, tras la ejecución de la instrucción protegida, a un estado que cumpla A2: {A1 " Bi} Si {A2}.

−*Llamada a una función*: cuando se produzca una llamada a una función especificada mediante pre y postcondición, y con el fin de facilitar el razonamiento, la escribiremos en una instrucción de asignación:

< x1, x2, ... ,xm> := f(e1, e2, ... ,ek)

debemos demostrar, justo antes de esta instrucción, que las expresiones que se pasen como argumentos cumplen la pre de la función; a continuación de esta asignación, las variables sobre las que se han asignado los resultados cumplirán la postcondición. Este cambio de nombre sigue los mismos criterios que las sustituciones de variables por expresiones.

Para:

funció f(p1,...., pk) ret r1,..., rm

{Pre: A1 (p1,..., pk)}

{Post: A2 (p1,..., pk, r1,..., rm)}

una llamada sobre los datos e1,..., ek que asigne los resultados sobre las variables x1,..., xm tendría pre y postcondición como sigue:

{