VERIFICACION DE PROGRAMAS

Uno de los enfoques para determinar si un programa es correcto es establecer una actividad de testing. Esta consiste en seleccionar un conjunto de datos de entrada para determinar si los resultados producidos por el programa con esos datos coinciden o no con los valores esperados. Para asegurar que el programa es correcto se debería analizar el mismo con todos los valores posibles de los datos de entrada. Pero esto es imposible cuando este conjunto es infinito. Por eso, el testing sólo puede mostrar la presencia de errores y no su ausencia.
Otro enfoque, la prueba formal de programas, es una técnica que se basa en el cálculo de predicados. Primero, se debe describir el comportamiento de cada instrucción del lenguaje formalmente. Es decir, se debe definir la semántica de un lenguaje de programación en términos de fórmulas lógicas. Para probar un programa, se debe expresar su semántica en términos de fórmulas lógicas y luego probar que el programa significa lo mismo que su especificación. Una prueba formal de un programa asegura que el programa es correcto con respecto a una especificación para todas las entradas. Hay dos problemas importantes con la prueba formal de programas: 1) la manipulación lógica puede ser tediosa y propensa a errores; 2) la prueba solamente muestra que el programa implementa la especificación correctamente. No hay certeza de que la especificación describe lo que el usuario realmente desea.

Definición: Si un programa usa n variables (x_1, x_2, ..., x_n) el estado **s** es una tupla de valores (X_1, X_2, ..., X_n) donde X_i es el valor de la variable x_i.
Ejemplo: Dado el estado s = (x, y) = (7, 8), el resultado de ejecutar la sentencia de asignación
x := 2*y+1 es el estado s'= (x, y) = (17, 8)

Una variable se usa en un programa para describir una posición de memoria que puede contener valores diferentes en diferentes estados. Una manera de describir un conjunto de estados es utilizando fórmulas del cálculo de predicados.

Definición: Sea U el conjunto de todos los posibles estados del programa y sea $S \subseteq U$. Se define P_S, el predicado característico de S, tal que $S = \{ s' \in S / P_S(s') \}$. Es decir, es el predicado que solamente es satisfecho por los estados que pertenecen al conjunto S.
Por ejemplo, en vez de decir que la sentencia x:= 2*y+1 transforma el estado s = (x, y) en el estado s'= (2*y+1, y), definiremos predicados P(x, y) y P'(x', y') tal que si P(x, y) es verdadero en el estado s entonces P'(x', y') será verdadero después de ejecutar la sentencia x := 2*y+1.

Ejemplo: Se quiere probar $x \leq 7$ después de ejecutar x := 2*y+1. Esto es verdadero si $2*y+1 \leq 7$ antes de ejecutar la sentencia, es decir $y \leq 3$.

$\{ y \leq 3 \}$ x := 2*y+1 $\{ x \leq 7 \}$

$\{y \leq 3\}$ y $\{x \leq 7\}$ se denominan aserciones.

Definición: Una aserción es una fórmula del cálculo de predicados ubicada en un programa, que es verdadera cada vez que el programa pasa por ese punto.

Definición: $\{P\}$ S $\{Q\}$, donde P y Q son aserciones llamadas pre-condición y post-condición respectivamente y S es un fragmento de programa, se interpreta como sigue: si la ejecución de S empieza en un estado caracterizado por P y S termina, entonces terminará en un estado caracterizado por Q.
S se dice parcialmente correcto con respecto a P y Q. Si además se garantiza que S termina, S es totalmente correcto con respecto a P y Q.
$\{P\}$ S $\{Q\}$ se denomina especificación formal de S.

Ejemplos:
Los siguientes son ejemplos de programas especificados en términos de pre y post condiciones.
1) Dados dos números naturales a y b, b ≠ 0, encontrar el cociente entre a y b.

P: { a, b ∈ N ∧ b > 0}

Q: { a = b*c + r ∧ 0 ≤ r < b ∧ c ≥ 0 }

2) Dado un arreglo de n números enteros, ordenarlo en forma ascendente.
P: {∀i: 1 ≤ i ≤ n: (integer(a[i]) ∧ a[i] = A[i]) } A referencia el valor de los elementos del arreglo a antes de iniciar la ejecución

Q: {∀i: 1 ≤ i < n: a[i] ≤ a[i+1] ∧ permutación(a, A)}
El predicado permutación(a, A) se define como:
permutación(a, A) = {∀i: 1 ≤ i ≤ n: (Nj: 1 ≤ j ≤ n: a[i] = a[j]) = (Nk: 1 ≤ k ≤ n: a[i] = A[k])}

3) Encontrar el valor máximo de un arreglo de n números enteros.
P: {∀i: 1 ≤ i ≤ n: (integer(a[i]) ∧ a[i] = A[i]) } A referencia el valor de los elementos del arreglo a antes de iniciar la ejecución

Q: {∃i: 1 ≤ i ≤ n: (max = a[i] ∧ ∀j: 1 ≤ j ≤ n: (a[j] ≤ max ∧ a[j] = A[j]))}

Volviendo al ejemplo { y ≤ 3 } x := 2*y+1 { x ≤ 7 }, { y ≤ 3 } no es la única pre-condición que hace verdadera la post-condición después de la ejecución de la sentencia de asignación. Otra pre-condición podría ser { y = 1 ∨ y = 3}. Pero esta última es menos interesante porque no caracteriza todos los estados desde los cuales se puede alcanzar un estado caracterizado por la post-condición.
Se desea determinar como pre-condiciones aquellas fórmulas que describan tantos estados como sea posible. Esto se hace eligiendo el predicado menos restrictivo posible.

Definición: Una fórmula A es más débil que una fórmula B si B → A (B es más fuerte que A). Como A es más débil que B, A describe más estados que B.
Una aserción es una fórmula del cálculo de predicados; como casos especiales se pueden distinguir las fórmulas T (true) y F(false). Como T es verdadero en todo estado, es la más débil entre las fórmulas. F es más fuerte que toda fórmula.
Por ejemplo, y ≤ 3 es más débil que (y = 1 ∨ y =3) porque (y = 1 ∨ y =3) → y ≤ 3

Definición: Para todo fragmento de programa S y fórmula Q, se define la pre-condición más débil R = wp(S, Q) tal que {R} S {Q} es verdadero. Es decir, wp(S, Q) representa todos los estados tal que la ejecución de S que comenzó en cualquiera de ellos, si termina, termina en un estado que satisface Q.
La notación {P} S {Q} (definida anteriormente) es otra notación para P → wp(S, Q).
wp se denomina también predicado transformador, ya que para cualquier fragmento de programa define una transformación de un predicado post-condición en un predicado pre-condición. Es decir, en vez de describir cómo un programa transforma un conjunto de estados iniciales en un conjunto de estados finales, describe cómo un programa transforma un predicado post-condición, que caracteriza el conjunto de estados finales, en un predicado pre-condición que caracteriza el conjunto de estados iniciales. Esto significa que para probar formalmente un programa se comenzará desde la post-condición y se trabajará “hacia atrás” para verificar cada una de las sentencias individuales del programa.

Se define a continuación el predicado wp para un lenguaje de programación con sentencias de asignación, condicional (if B S_1 else S_2) e iteración (while B S).

Definición: wp(x = e, Q) = Q {x/e} e es una expresión

Por ejemplo

wp(y = b, y ≥ 0} = (y ≥ 0) {y/b} = b ≥ 0

wp(x = x+1, x = 5) = (x = 5) {x/x+1} = (x+1 = 5) = (x = 4)

Definición: wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))

Por ejemplo se desea calcular la pre-condición más débil del siguiente fragmento de programa:

```
x = x+1;
y = y+2
{x < y}
```

wp(x = x+1; y = y+2, x < y) = wp(x = x+1, wp(y = y+2, x < y))
= wp(x = x+1, x < y+2)
= x < y+1

Esto se puede escribir también como sigue

```
{ x < y+1 }
x = x+1;
{ x < y+2 }
y = y+2
{ x < y }
```

Definición: wp(if B S_1 else S_2, Q) = (B → wp(S_1, Q)) ∧ (¬B → wp(S_2, Q))

Ejemplo: Calcular la pre-condición más débil del siguiente fragmento de programa:

```
if y = 0
  x = 0
else
  x = y – 1
{ x = y }
```

wp(if y = 0 x = 0 else x = y-1, x = y) =
= (y = 0 → wp(x = 0, x = y)) ∧ (¬(y = 0) → wp(x = y-1, x = y))
= (y = 0 → y = 0) ∧ (y ≠ 0 → y-1 = y)
= T ∧ (y ≠ 0 → F)
= (y ≠ 0 → F)
= (¬(y ≠ 0) ∨ F)
= ¬(y ≠ 0)
= (y = 0)
Por lo tanto la pre-condición más débil es P: { y = 0 }

Definición: Un predicado I se llama invariante de una sentencia S si wp(S, I) = I

Definición: Para probar la corrección de una sentencia while B S, con post-condición Q se deberá:
1) Calcular el predicado invariante I.

2) Probar que $I \wedge B \rightarrow wp(S, I)$
3) Probar que I vale antes de iterar
4) Probar que $I \wedge \neg B \rightarrow Q$ (Q es la post-condición de la sentencia while B S)

Ejemplo:
Para el siguiente fragmento de programa y sus pre y post condiciones, probar si es o no correcto con respecto a las especificaciones dadas.

pre: {n >0}
(6)
i = 1;
(5)
suma = 0;
(4)
while (i <= n)
(3)
{ suma = suma + i;
(2)
i++;
(1)
}

post: $\{suma = \sum_{j=1}^{n} j\}$

El predicado invariante es I: $\{ 1 \leq i \leq n+1 \wedge suma = \sum_{j=1}^{i-1} j \}$

(1) = I (es decir, después de ejecutar i++ es verdadero el predicado invariante)

(2) = (1) $\{i/i+1\} = \{1 \leq i+1 \leq n+1 \wedge suma = \sum_{j=1}^{i+1-1} j\} = \{0 \leq i \leq n \wedge suma = \sum_{j=1}^{i} j\}$

(3) = (2)$\{suma/suma+i\} = \{0 \leq i \leq n \wedge suma+i = \sum_{j=1}^{i} j\} = \{0 \leq i \leq n \wedge suma+i = \sum_{j=1}^{i-1} j + i\}$

(3) = $\{0 \leq i \leq n \wedge suma = \sum_{j=1}^{i-1} j\}$

$I \wedge \{i <= n\} \rightarrow (3)$

$\{1 \leq i \leq n+1 \wedge suma = \sum_{j=1}^{i-1} j\} \wedge \{i <= n\} \rightarrow (3)$

$\{1 \leq i \leq n \wedge suma = \sum_{j=1}^{i-1} j\} \rightarrow \{0 \leq i \leq n \wedge suma = \sum_{j=1}^{i-1} j\}$

$I \wedge \{i > n\} \rightarrow post$

$\{1 \leq i \leq n+1 \wedge suma = \sum_{j=1}^{i-1} j\} \wedge \{i > n\} \rightarrow post$

$\{ i = n+1 \wedge suma = \sum_{j=1}^{i-1} j \} \rightarrow \{ suma = \sum_{j=1}^{i-1} j \}$

$\{ suma = \sum_{j=1}^{n} j \} \rightarrow \{ suma = \sum_{j=1}^{n} j \}$

(4) = I

$(5) = (4)\{suma/0\} = \{ 1 \leq i \leq n+1 \wedge 0 = \sum_{j=1}^{i-1} j \}$

$(6) = (5)\{i/1\} = \{ 1 \leq 1 \leq n+1 \wedge 0 = \sum_{j=1}^{1-1} j \} = \{ 0 \leq n \wedge 0 = 0 \} = \{ 0 \leq n \}$

La aserción (6) es la pre-condición más débil del programa.
Se debe comprobar además si la pre-condición del programa es más fuerte que la pre-condición más débil del programa, esto es

$$\{ n > 0 \} \rightarrow \{ n \geq 0 \}$$

Es inmediato que se cumple.

Por lo tanto, se ha probado formalmente que el fragmento de programa dado es parcialmente correcto. Para probar que es totalmente correcto se debería probar que el programa termina y para probar esto se debería probar que el único ciclo que tiene el programa termina.