CHAPTER 1

# Escribiendo Programas correctamente

## 1. Introducción

Una de las tareas mas importantes de la programación es el de verificar la corrección de los programas. Durante muchos años se han prometido programas que podrían verificar a otros. Desde la década de los 60 seguimos esperando. Aún no existe esta posibilidad.

Hoy en día se hace mucho mas importante la verificación de los programas. Existen programas en diferentes industrias que en caso de fallas pueden causar inclusive la muerte de personas y otros que no permiten pruebas experimentales.

Como ejemplo podemos mencionar por ejemplo los programas que controlan el vuelo de los aviones, servo mecanismos asistidos por computadora e inclusive muchos de los electro domésticos que existen en su hogar.

En este tiempo se tiene un conocimiento más profundo de la programación y se pueden realizar muchas mas verificaciones que solo las de la caja negra que pueden indicarnos bien o mal.

El rol el control de calidad y la verificación de programas nos permite encontrar errores de los programas. La codificación es una de las habilidades requeridas para esto. Esta temática es la que se trata en este capitulo.

## 2. Especificación de programas

No solo puede estar un programa con errores, también puede estar equivocada la especificación del mismo. Por esta razón también debe verificarse las especificaciones de los mismos. Aunque no es parte del texto un tratamiento riguroso de la verificación y prueba de programas es necesario tratar este tema que se estudia en los cursos de métodos formales.

Llamemos a $\{P\}$ precondición, a $\{Q\}$ post condicion y $C$ un programa la notación

$$\{P\}C\{Q\}$$

significa que si partimos de un estado $P$ después de que el programa termine llegaremos al estado $Q$. Esta notación se debe a C.A.R. Hoare. Ejemplo:

$$\{X = 1\}X = X + 1\{X = 2\}$$

Este pequeño código especifica que si $X$ es inicialmente 1 una vez corrido el programa $X$ toma el valor de 2 que es claramente verdadero.

Una expresión del tipo $\{P\}C\{Q\}$ es una especificación de correctitud parcial porque para verificar que está correcta no es necesario que el programa $C$ termine.

Una especificación mas fuerte es la especificación de correctitud total en la que se pide la prueba de que el programa termine. La notación que se utiliza para esto es:

$$[P]C[Q]$$

la relación que existe entre ambas definiciones es:

$$\textit{correctitud total} = \textit{terminacion} + \textit{correctitud parcial}$$

Ejemplo:

$$[X = x \wedge Y = y]\ R = X;\ X = Y;\ Y = R\ [X = y \wedge Y = x]$$

Especifica que una vez que se procese el código los valores de $X, Y$ se intercambian.
Ejemplo:

$$\begin{aligned}
&\{Y = y\}\\
&R = X;\\
&Q = 0;\\
&WHILE\ Y \leq R\\
&\ R = R - Y; Q = Q + 1\\
&\{R < y \wedge X = R + (YxQ)\}
\end{aligned}$$

Este ejemplo es menos intuitivo para la verificación pero nos indica claramente que el resultado de $X$ es independiente del valor inicial. También hay que hacer notar que la variable $Q$ del programa que no esta especificada en la precondición y post condición por lo que no se puede decir nada al respecto.

Para el estudio de esta temática existe toda una lógica desarrollada por R.S. Floyd y Hoare denominada lógica de *Floyd - Hoare*

**2.1. Porque especificar.** La primera pregunta que uno se hace es *porque especificar*. Uno podría especificar para tener mayor documentación del sistema, desea una descripción mas abstracta, o porque desea realizar un análisis del mismo.

Lo que uno debe preguntarse es porque especificar *formalmente* y las respuestas que se pueden conseguir de desarrolladores profesionales son:

(1) Mostrar que una propiedad se mantiene globalmente en un programa
   - Quiero caracterizar la condición de correctitud.

- Quiero mostrar que la propiedad es realmente una invariante
- Quiero mostrar que mi sistema cumple algún criterio de alto nivel en el diseño

(2) Manejo de errores
- Quiero especificar que ocurre cuando se produce un error
- Quiero especificar que las cosas correctas ocurren cuando se produce un error
- Quiero estar seguro que no exista este error

(3) Completitud
- Quiero estar de seguro que he cubierto todos los casos, incluyendo los casos de error para un protocolo
- Quisiera saber si el lenguaje que he diseñado es computacionalmente completo

(4) Especificar Interfaces
- quisiera definir una jerarquia de clases de $C++$
- quisiera una descripción mas formal de la interfase del usuario

(5) Manejar la complejidad
- El diseño esta muy complicado para manejar todo en la cabeza y necesitamos una forma de pensar en pedazos mas pequeños

(6) Cambiar el Control
- Cada vez que cambio un pedazo de código quisiera conocer que otras partes son afectadas sin mirar todos los módulos y sin mirar el código fuente.

**2.2. Que especificar.** Los métodos formales no están desarrollados de tal forma que un sistema entero y muy grande pueda ser especificado completamente. Solo se pueden especificar algunos aspectos tales como funcionalidad y comportamiento de un sistema de tiempo real.

AL escribir una especificación un debe decidir si la descripción es requerida y permitida. ¿Puede o debe? Una vez que uno tiene claro lo que hay que especificar un puede determinar que puede formalizarse.

2.2.1. *Condiciones de Correctitud.* Esto implica la noción de una condición global de correctitud que el sistema debe mantener. También puede ser algo standard coherencia del cache, ausencia de bloqueos mutuos.

2.2.2. *Invariantes.* La forma de caracterizar algunas condiciones que no cambian de un estado a otro se denominan *invariantes.*

2.2.3. *Comportamiento Observable.* Las invariantes son una buena forma de caracterizar las propiedades deseadas de un sistema. Formalizar las transiciones le permite probar que se mantienen. Cuando

se especifican las transiciones de estado se esta especificando el comportamiento del sistema cuando interactua con su ambiente, que es el *comportamiento observable*

2.2.4. *Propiedades de entidades.* Las propiedades mas importantes de las entidades se expresan comúnmente por el tipo. Sin embargo para entidades estructuradas podemos ver otras propiedades de las que podemos anotar por ejemplo

- Orden. ¿Los elementos están ordenados?
- Duplicados. ¿Se Permiten?
- Rangos. ¿Puede acotarse los elementos de alguna manera?
- Acceso asociativo. ¿Los elementos se recuperan por indices o llaves?
- Forma ¿El objeto tiene una estructura lineal,jerarquia, arbitraria, gráfica, etc?

**2.3. Como Especificar.** Dado que entiende lo que desea especificar y lo que desea puede empezar a especificar. La técnica principal es la *descomposición* y *abstracción.* Para esto damos algunas ideas:

- Abstraer. No piense como programador.
- Piense en función de la definición no la funcionalidad
- Trate de construir teorías no solo modelos
- No piense en forma computacional

**2.4. Como proceder incrementalmente.** En cada nivel de abstracción ignoramos muchos detalles sobre el sistema mas abajo. Uno puede estar deseoso de especificar todo desde el principio con el temor de tener muchas especificaciones incompletas para lo cual se puede indicar:

(1) Asuma que algo es verdadero en la entrada y capture estas presunciones en las precondiciones
(2) Primero considere el caso normal y después el caso de error
(3) Primero ignoré algunos hechos tales como si esta ordenado, tiene duplicados y posteriormente fortalezca la post condición.
(4) Primero asuma que las operaciones son atómicas y luego divida estas en otras mas pequeñas.

**2.5. Invariantes.** Definamos con mas claridad una invariante. Una invariante es una propiedad qu es verdadera y no cambia durante le ejecucion del mismo. Esta propiedad engloba a la mayor parte de los elementos o variables del programa. Cada ciclo tiene una propiedad invariantes. Las invariantes explican las estructuras de datos y algoritmos. Son útiles para tareas de mantenimiento y diseño.

Las propiedades de las invariantes deben mantenerse cuando se modifica el programa. La invariante obtenida se utiliza en un proceso estatico para verificar la corectitud del programa.

2.5.1. *Utilizacion de las invariantes.* Las invariantes sonutilies en todos los aspectos de la programación: diseño, codificacion, prueba, mantenimiento y optimizacion. Presentamos una lista de usos a fin de motivar a la utilizacion de las mismas por los programadores.

- Escribir mejores programas.- Muchos autores coinciden que se pueden construir mejores programas cuando se utilizan en el diseño de los programas. Las invariantes formalizan en forma precisa el contato de un trozo de codigo. El uso de invariantes informalmente tanmbien ayuda a llos programadores.
- Documentar el programa.- Las invariantes caracteriza ciertos aspectos de la ejecucion del programa y proveen una documentacion valiosa sobre el algoritmo, estructura de datos y operacion. Ellas proveen un nivel de conocimientos a un nivel necesario para modificar el mismo.
- Verificar las supociciones.- Teniendo las invariantes las podemos utilizar en la verificacion del progrma observando que no cambian a medida que se procesa.
- Evitar errores.- Las invariantes pueden prevenir que el programa realice cambios que acidentalmente cambien las suposiciones bajo las cuales su funcionamiento es correcto.
- Formar un espectro.- El espectro de un programa son propiedades medibles en el programa o su ejecución. Algunas de estas son el numero de lineas ejecutadas por el prgrama, el tamaño de la salida o propiedades estaticas como la complegidad ciclomatica. La diferencia entre varios especrtros del nos muestra cambios en los datos o el programa.
- Ubicar condiciones inusuales.- Condiciones inusuales o excepcionales deben llamar la atencion a los programadores porque pueden ser causa de posibles errores.
- Crear datos de prueba.- Las invariantes pueden ayudar en la generacion de datos de prueba de dos maneras. Pueden infringir intencionalmente las invariantes ampliando los datos de prueba. La otra es de mantener las invariantes para caracterizar el uso correcto del programa.
- Pruebas.- Prueba de teoremas, analisis de flujo de datos chequeo del modelo, y otros mecanismos automatizaods o semi automatizados pueden verificar la correctitud del programa con respecto a su especificacion.

2.5.2. *Ejemplos.*

(1) Mostrar que la invariante es $PX^N = x^n \wedge x \neq 0$

```
P:=1;
IF not (x=0)
    while not (N=0)
        IF IMPAR(N) P=PxX
```

```
        N=N /2
        X=X x X
ELSE
    P=0
```

aquí la precondición es $\{X = x \wedge N = n\}$ y la post condición $\{P = x^n\}$. Veamos varios detalles del programa:

- tiene tres variables $P, N, X$
- Los valores iniciales son un valor $x, n$ cualesquiera, precondición
- La post condición indica que el resultado es $x^n$

Demostrar que $PX^n$ es una invariante. Supongamos que $X = 2$ y $N = 9$

| Iteración | N | P | X | $PX^N$ |
|---|---|---|---|---|
| 1 | 9 | 1 | 2 | 512 |
| - | 9 | 2 | 2 | – |
| - | 4 | 2 | 4 | 512 |
| 2 | 2 | 2 | 4 | – |
| - | 2 | 2 | 16 | 512 |
| 3 | 1 | 2 | 16 | – |
| - | 1 | 2 | 256 | 512 |
| 4 | 1 | 512 | 256 | 512 |
| - | 0 | 512 | 256 | – |
| - | 0 | 512 | 65536 | – |

Como ve hemos anotado todos los valores que toman las variables involucradas y anotado los valores que toman y al final de cada iteración calculado la supuesta invariante y comprobamos que es correcta porque sus valores no cambian.

(2) Las invariantes pueden tomar diferentes formas no solamente la vista, consideremos por ejemplo el siguiente programa
pre condicion $\{M \geq 1\}$

```
X=0;
for N=1 to M
    X=X+M
```

post condicion $\{X = (M(M-1))/2\}$

Para este algoritmo podemos utilizar una invariante que es una sumatoria

$$X = \sum_{i=0}^{N-1} i$$

En este ejemplo vemos que efectivamente $X$ en cualquier momento representa el resultado de la sumatoria hasta el numero de iteracion en que se encuentra.

La postcondion puede mostrarse que se da hallando el resultado de la suma. En este caso $X$ varia en cada iteracion

pero $M$ se mantiene constante por lo que no puede ser parte de la invariante.

Lo que queda por aclarar es que se ha tomado $N-1$ porque cuando termina el ciclo el valor de $N$ es uno mas que el ultimos.

Cuando tenemos varios ciclos tenemos que escribir una invariante por cada ciclo.

Como se ve en la invariante participan las variables utilizadas en la solución. Con estas se puede realizar una prueba formal (matemática) con el uso de la lógica para probar la correctitud del programa, pero esta temática excede los alcances del texto.

## 3. Utilizando principios de verificación para construir programas

Supongamos que queremos escribir in programa de búsqueda binaria escribimos una especificación. Primero debemos determinar si el arreglo $[0..n-1]$ esta ordenado. Conocemos con presicion que $n \geq 0$ y que $x[0] \leq x[1] \leq x[2] \leq ... \leq x[n-1]$.

El pseudo código debe funcionar tanto para enteros, cadenas, reales siempre y cuanto todos los elementos sean del mismo tipo. La respuesta esta almacenada en un entero $p$ que representa la posición en el arreglo donde se encuentra un elemento $t$ que estamos buscando $-1$ indica que el elemento buscado no existe en el arreglo.

La búsqueda binaria resuelve el programa recordando permanentemente el rango en el cual el arreglo almacena $t$. Inicialmente el rango es todo el arreglo y luego es reducido comparando $t$ con el valor del medio y descartando una mitad. El proceso termina cuando se halla el valor de $t$ o el rango es vació. En una tabla de $n$ elementos toma $\log_2 n$ comparaciones.

**3.1. Escribiendo el programa.** La idea principal es que $t$ debe estar en el rango del vector. Podemos utilizar la descripción para construir el programa.

```
inicializar el rango entre $0..n-1$
loop
    {invariante: debe estar en el rango}
    si el rango es vació
        terminar y avisar que t no esta en el arreglo
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

La parte esencial del programa es la invariante que al principio y final de cada iteración nos permite tener el estado del programa y formalizar la noción intuitiva que teníamos.

Ahora refinaremos el programa haciendo que todas las acciones que se tomen respeten la invariante del mismo.

Primero buscaremos una representación del rango digamos $l..u$ entonces la invariante es *el rango debe estar entre* $l..u$. El próximo paso es la inicialización y debemos estar seguros que respeten la invariante, la elección obvia es $l = 0, u = n - 1$.

Trasladando esto al programa

```
l=0; u=n-1
loop
    {invariante: debe estar en el rango l,u}
si el rango es vació
    terminar y avisar que t no esta en el arreglo
calcular m que es el medio del rango
use m con una prueba para reducir el rango
si se encuentra t en el proceso de reducción
    terminar y comunicar su posición
```

Continuando con el programa vemos que el rango es vació cuando $l > u$. en esta situación terminamos y devolvemos $p = -1$. Llevamos esto al programa tenemos

```
l=0; u=n-1
loop
    {invariante: debe estar en el rango l,u}
    if l > u
    p=-1; break;
calcular m que es el medio del rango
use m con una prueba para reducir el rango
si se encuentra t en el proceso de reducción
    terminar y comunicar su posición
```

Ahora calculamos $m$ con $m = (l+u)/2$ donde / implementa la division entera. Las siguientes líneas implican el comparar $t$ con $x[m]$ en la que hay tres posibilidades, por igual, menor y mayor. Con lo que el programa queda cono sigue

```
l=0; u=n-1
loop
    {invariante: debe estar en el rango l,u}
    if l > u
    p=-1; break;
    m=(l+u)/2;
    case
        x[m]< t : l=m+1;
        x[m]= t : p=m: break;
        x[m]> t : u=m-1
```

Normalmente un análisis formal para verificar el programa se hace de abajo hacia arriba.

El principio de correctitud se basa en

- *inicialización* La invariante es verdadera cuando se ejecuta el programa por primera vez.
- *preservación* la invariante se preserva al principio, al final de cada uno de los ciclos del programa y cuando este termina.
- *terminación* El programa termina y produce los resultados esperados. Esto puede probarse utilizando los hechos que hacen la invariante.

Para comprobar la ultima parte de la especificación que indica que el programa toma $\log_2 n$ comparaciones podemos reescribir este programa en forma recursiva:

```
l=0; u=n-1
procedure buscar(l,u)
    {invariante: debe estar en el rango l,u}
    if l > u
    p=-1; return p;
    m=(l+u)/2;
    case
        x[m]< t : l=m+1: buscar(l,u)
        x[m]= t : p=m: return m;
        x[m]> t : u=m-1: buscar(l,u)
```

de este programa vemos que

$$T(n) = \begin{cases} 1 & l > u, \\ T(n/2) + 1 & \text{otros casos.} \end{cases}$$

de donde resolviendo esta recursión hallamos el que el numero de comparaciones es lo pedido.