Unidad II

Verificación de Software

Conceptos

- Validación. ¿Estamos fabricando el producto correcto? Programa → usuario
- Verificación. ¿Estamos fabricando correctamente el producto? Programa → especificación
- Todas las actividades que se emprenden para asegurar que el software cumple sus objetivos”
- Se deduce que tanto el proceso como el producto deben ser V&V.

Conceptos (2)

- Llevar a cabo la V&V sólo luego de que se dispone del código final, hace que sea muy difícil y costoso reparar los errores detectados.
- Todo debe ser V&V: cualidades, subproductos, proceso, la misma V&V, etc.
- Los resultados pueden no ser binarios:
  - La presencia de defectos en el soft industrial no puede evitarse por completo en la práctica
  - En ocasiones algunos defectos pueden tolerarse
  - En la práctica, la corrección es relativa
- Aun las cualidades implícitas deben V&V.

Técnicas de V&V

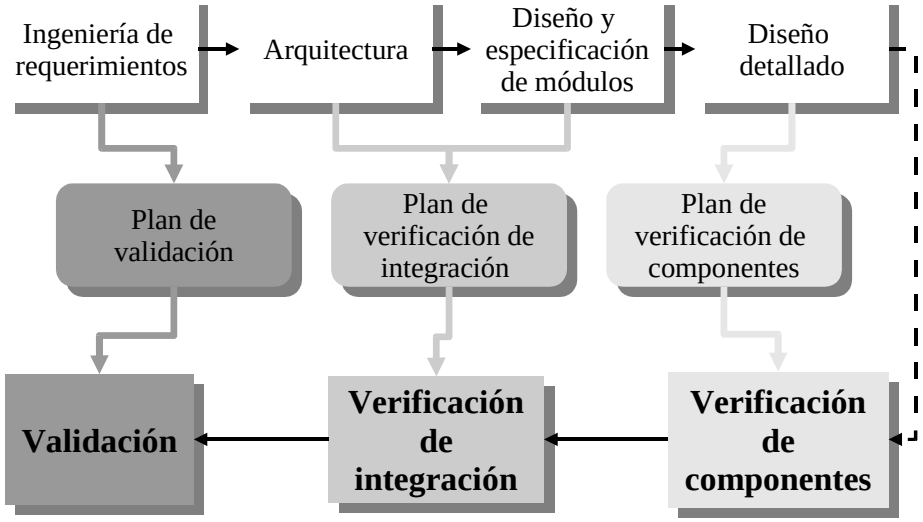
- Verificación formal de programas y modelos
- Testing
- Simulación
- Chequeo de modelos (*model checking*)

- Construcción formal de programas

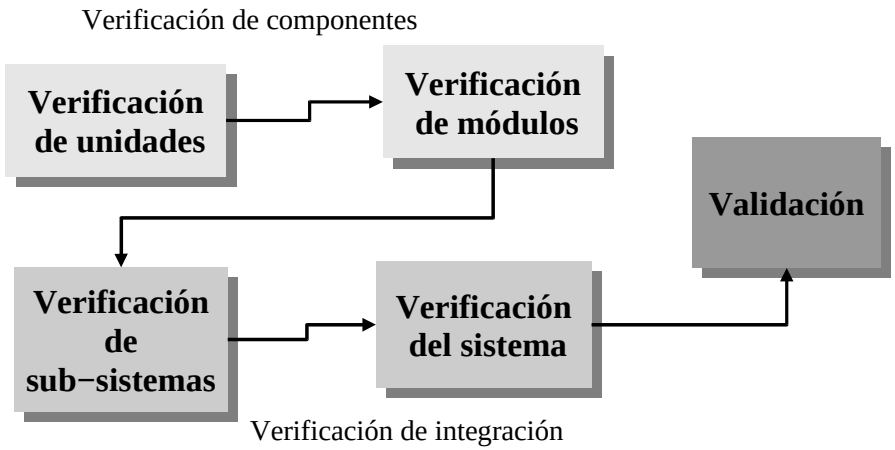
Técnicas de V&V (2)

- Poner a prueba un programa puede usarse para mostrar la presencia de errores, pero nunca para mostrar su ausencia.
- Las técnicas estáticas pueden usarse sólo para chequear la correspondencia entre el soft y la especificación; no pueden probar que el soft es operacionalmente correcto.
- Verificación de modelos es una técnica automática para verificar sistemas reactivos modelados con FSM y especificados con fórmulas de una lógica temporal proposicional.

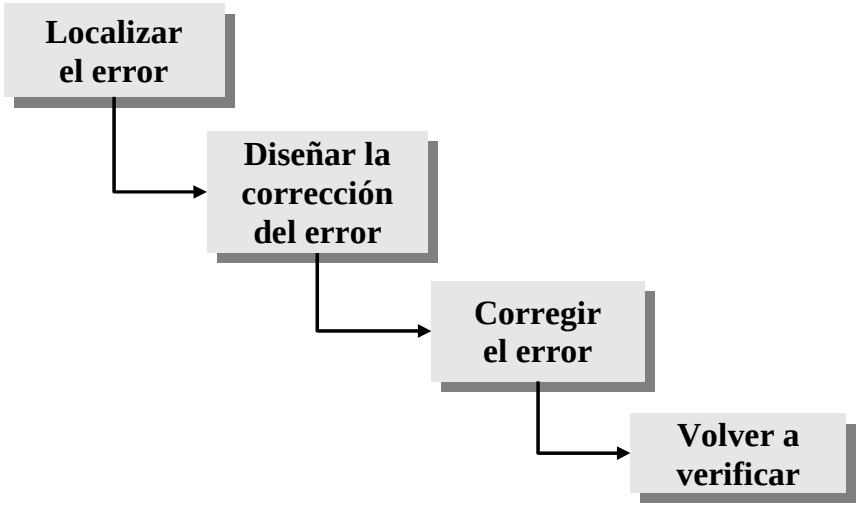
Planificación de V&V (2)



Planificación de V&V



Planificación de V&V (3)



Testing estructural

Vocabulario

- Llamamos *dominio* de un programa al producto cartesiano entre los tipos de datos de las variables de entrada.
  - ¿Funciones parciales o totales?
- Dado un programa *P* con dominio *D* un *caso de prueba* para *P* es un elemento *d* perteneciente a *D*.
- Dado un programa *P* con dominio *D* un *conjunto de casos de prueba* (o conjunto de prueba) para *P* es cualquier subconjunto de *D*.

Vocabulario (2)

- Dado un programa *P* y un caso de prueba *d* para *P*, decimos que *P* es correcto en *d* si *P(d)* es la salida esperada para *P* cuando ejecuta *d*.
- Dado un programa *P* y un conjunto de prueba *T*, decimos que *P* es correcto en *T* si *P* es correcto para todo *d* en *T*.
- Un *criterio* o *táctica de selección de pruebas* (criterio de prueba, criterio o táctica) es una regla, procedimiento o algoritmo que permite calcular conjuntos de prueba para cualquier programa.

Vocabulario (3)

- Los criterios se basan en dividir el dominio de un programa en *clases de equivalencia*, D_i, con la esperanza de que el programa se comporte igual para todo *d* en D_i.
- Un criterio *C* es más fino que otro criterio *C'* sii para todo programa *P* y todo conjunto de prueba *T* de *P* que satisface *C*, existe un subconjunto de *T* que satisface *C'* para *P*.

Testing estructural

- Esta forma de testing utiliza la información sobre la estructura interna del programa.
- Testing de caja blanca.
- Se prueba lo que el programa *hace* y no lo que *se supone que hace*.
- Un criterio para testing estructural se define indicando de forma genérica un subconjunto de caminos que deben recorrerse.
  - Cualquier conjunto de prueba que logre recorrer todos esos caminos satisface el criterio.

Testing estructural (2)

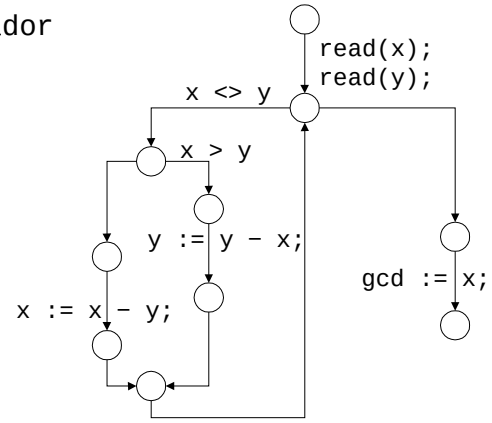
- Puede automatizarse en gran medida.
- Requiere que haya finalizado la fase de codificación para que pueda empezarse a testear.
- Si se cambia la estructura del código deben re−calcularse los casos.
- La fase más compleja es determinar los valores particulares de las variables de entrada de forma tal que al ejecutar el programa se recorra cierto camino particular (indicado por el criterio).

Cubrimiento de sentencias

- Seleccionar *T* tal que, al ejecutar *P* para cada *d* en *T*, cada sentencia elemental de *P* es ejecutada al menos una vez.
- Un error no puede descubrirse si la parte donde está no es probada.
- No siempre que se ejecute una sentencia esta provocará la falla.
- ¿Sentencia?
- Minimizar *T*.

Ejemplo: Euclides

```
//Máximo común denominador
begin
read(x);
read(y);
while x <> y loop
    if x > y then
        x := x − y;
    else
        y := y − x;
    endif;
endloop;
gcd := x;
end;
```



Ejemplo: Euclides

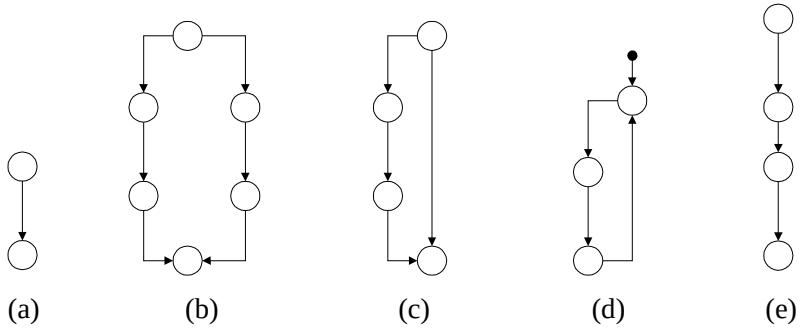
Para probar el algoritmo de Euclides según el criterio de cubrimiento de sentencias es suficiente el conjunto de prueba:

```
T = {<x := 3, y := 2>}
```

Aunque muchos otros también satisfacen el criterio. Lo complejo es determinar que 3 y 2 son los valores de entrada que hacen que el programa utilice todas sus sentencias.

¿Funciona bien el programa cuando `y > x`? ¿Y cuando `x = y`? Al probar únicamente con T, ¿qué se está probando? ¿Qué suposiciones estamos haciendo?

Grafo de flujo de control (2)

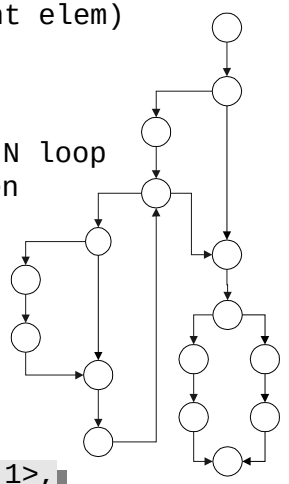


Grafo de flujo de control (GFC)

- Las sentencias se representan con flechas; los nodos indican los puntos de entrada y salida de la sentencia.
- El grafo se arma inductivamente así:
  - Por cada sentencia de E/S, asignación o llamada a procedimiento, se genera un grafo tipo (a).
  - Sean *S1* y *S2* dos sentencias y G1 y G2 sus correspondientes grafos entonces:
    - `if` *cond* `then` *S1* `else` *S2* `endif` corresponde a (b)
    - `if` *cond* `then` *S1* `endif` corresponde a (c)
    - `while` *cond* `loop` *S1* `endloop` corresponde a (d)
    - *S1*`;` *S2* corresponde a (e)

Ejemplo: búsqueda lineal

```
int linsearch(int table[], int N, int elem)
found := false;
if N <> 0 then
    counter := 1;
    while (not found) and counter < N loop
        if table[counter] = elem then
            found := true;
        endif
        counter := counter + 1;
    endloop
endif;
return found?counter – 1:0;
end;
```



```
T = {<table:=[1,2], N := 2, elem := 1>,
     <table:=[1,2], N := 2, elem := 3>}
```

Cubrimiento de flechas

- Seleccionar *T* tal que, al ejecutar *P* para cada *d* en *T*, cada flecha del GFC de *P* es atravezada al menos una vez.
- Se prueban las condiciones y las sentencias, aunque las condiciones no son analizadas ni se tienen en cuenta las relaciones entre las condiciones de cada sentencia.
- Es más fino y produce resultados diferentes que el criterio de cubrimiento de sentencias.

Cubrimiento de condiciones

- Seleccionar *T* tal que, al ejecutar *P* para cada *d* en *T*, cada flecha del GFC de *P* es atravezada y todos los posibles valores de las proposiciones simples que componen las condiciones son probados al menos una vez.
- Se analizan parcialmente y prueban las condiciones y se prueban las sentencias, pero no se tienen en cuenta las relaciones entre las condiciones de cada sentencia.
- Es más fino y produce resultados diferentes que el criterio de cubrimiento de flechas.

Ejemplo: búsqueda lineal (2)

Para cumplir con cubrimiento de flechas falta un caso:

```
T = {<table :=[1,2], N := 2, elem := 1>,
     <table :=[1,2], N := 2, elem := 3>,
     <table :=[],    N := 0, elem := 1>}
```

El conjunto anterior también verifica cubrimiento de condiciones.

Aun así no es posible detectar el error en

```
counter < N
```

Más criterios

- Existen otros criterios basados en flujo de control que verán en la práctica (uno de ellos mejora el cubrimiento de condiciones).
- Además, hay criterios basados en flujo de datos: rastrean los valores asignados a las variables de entrada a medida que son modificados y transferidos a otras variables, hasta que son utilizados para producir los resultados.
- Ver el paper de Rapps y Weyuker.
- No es simple testear cuando hay arreglos o punteros.
  - Tesis de grado Dacharry, Bertoni y Peñaranda

Testing estructural (3)

- En general, nunca se aclara cómo hacer para saber si un caso de prueba descubrió un error o no.
- Obviamente, eso lo dice la especificación.
- Sin especificación es imposible testear, pues la corrección se define con respecto a la especificación.
- Para algunos la especificación está tan ausente o lejana que la suelen llamar *oráculo*.
  - En muchos casos el oráculo es el *cliente*.
- En muchos casos el testing es una farsa.

El ciclo del testing