

**El testing como parte del
proceso de calidad del software**

Introducción al análisis estático

Contenido



- ?

Análisis estático

- Analiza un programa sin ejecutarlo.
- No depende de tener buenos casos de prueba
 - incluso ningún caso de prueba.
- No conoce lo que se supone hace el programa.
 - Busca violaciones a reglas de programación razonables.
- No es un reemplazo del testing
 - Ayuda a encontrar problemas en caminos no probados
 - Pero muchos defectos no pueden ser encontrados con análisis estático

Prejuicios comunes

- Los programadores son inteligentes.
- La gente inteligente no comete errores tontos.
- Usamos buenas técnicas para detectar errores de manera temprana (testing unitario, revisión de pares, inspección de código...)
- Los errores que permanecen son sutiles así que las técnicas de análisis estático tienen que ser muy sofisticadas para encontrarlos.

Prejuicios comunes

- ¿Cuál es el error?

```
if (listeners == null)  
    listeners.remove(listener)
```

- JDK1.6.0, b105, sun.awt.x11.XMSelection
(líneas 243-244)

¿Por qué existen los defectos?

- Nadie es perfecto
- Tipos comunes de errores:
 - Mala comprensión de las características del lenguaje, y los métodos de la API.
 - Errores al escribir (operador booleano equivocado, falta de paréntesis, etc.).
 - Mala comprensión de una clase y su invariante.
- Todo el mundo comete errores de sintaxis, pero el compilador los detecta.
 - ¿Qué sucede con los errores que no son sintácticos?

Un problema teórico

- Escribimos un programa
 - y queremos saber si cumple cierta propiedad (por ejemplo que no dereferencia un puntero Null, no hay división por cero, etc.)
- Verificar la propiedad manualmente es impráctico en cualquier caso real.
- Parece buena idea escribir una herramienta de análisis estático que verifique la propiedad.

Un problema teórico

- Pero es imposible escribir ese programa!
- El problema es *indecidable* (en general).
 - Teorema de Rice: para cualquier propiedad no trivial, no existe ningún método automático que pueda determinar si tal propiedad se satisface en un programa arbitrario.
 - No trivial significa que existe un programa que la satisface y otro que no.
- Entonces, ¿nos damos por vencidos?

Un problema teórico

- Solo si nos molesta no tener una respuesta absolutamente precisa.
- Truco
 - Abstractar el comportamiento del programa (sobre o subaproximación).
 - Probar la propiedad sobre el programa abstracto.
- Aún así podemos tener buenas garantías sobre el programa.

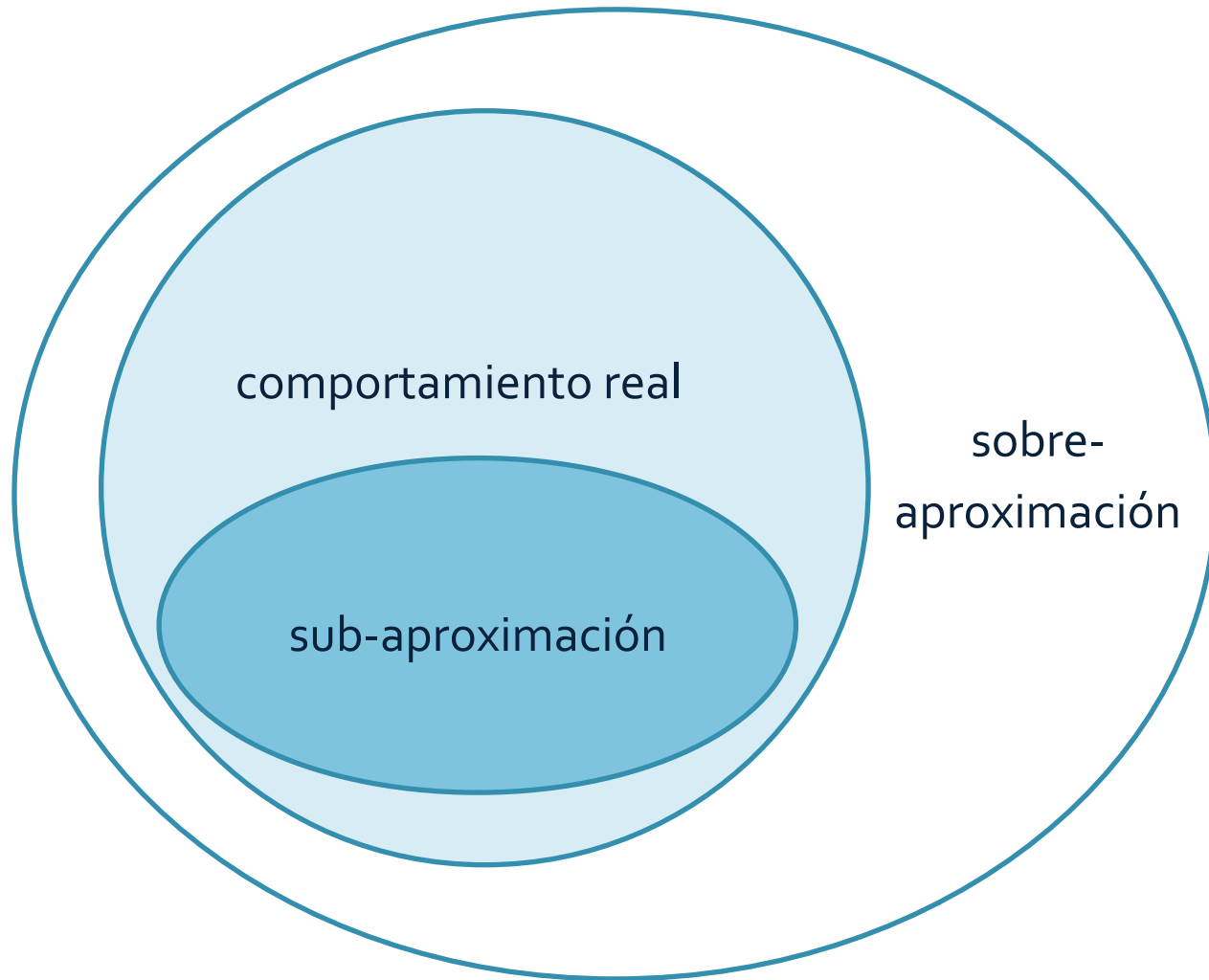
Opciones para la abstracción

- **Análisis estático correcto:**
 - Sobre-aproxima el comportamiento del programa.
 - Garantiza la detección de todas las violaciones de la propiedad.
 - Pero puede reportar “falsos positivos”.
- **Análisis estático completo:**
 - Sub-aproxima el comportamiento del programa.
 - Cualquier violación reportada es efectivamente una violación a la propiedad
 - Pero no se garantiza que todas las violaciones sean reportadas.

Opciones para la abstracción

- No puede existir un análisis estático correcto y completo.
- Cuando un análisis correcto no reporta violaciones, entonces efectivamente no ocurre ninguna.
 - Es una garantía fuerte.
 - La mayor parte de los análisis optan por ser correctos en lugar de completos.

Complejidad y corrección



Ejemplo: análisis de signo

- La abstracción $[[n]]$ transforma un número n en signo o cero:
 - $[[n]] = +$ cuando n es positivo
 - $[[n]] = -$ cuando n es negativo
 - $[[0]] = 0$

Ejemplo: análisis de signo

- La abstracción $[[\odot]]$ transforma cualquier operador $\odot (+, \times, /)$ en un operador entre signos:
 - $+ [[+]] \circ = +$
 - $+ [[+]] + = +$
 - $+ [[+]] - = \text{"incierto"}$
 - $\text{"incierto"} [[+]] + = \text{"incierto"}$
 - ...

Ejemplo: análisis de signo

- La abstracción $[[\odot]]$ transforma cualquier operador $\odot (+, \times, -, /)$ en un operador entre signos:
 - $+ [[x]] \text{ o } 0 = 0$
 - $+ [[x]] + = +$
 - $+ [[x]] - = -$
 - “incierto” $[[x]] + = \text{“incierto”}$
 - ...

Ejemplo: análisis de signo

- La abstracción $[[\odot]]$ transforma cualquier operador $\odot (+, \times, -, /)$ en un operador entre signos:
 - $+ [[/]] o = \text{"indefinido"}$
 - $+ [[/]] + = +$
 - $+ [[/]] - = -$
 - $+ [[/]] \text{"incierto"} = \text{"indefinido"}$
 - $\text{"incierto"} [[/]] o = \text{"indefinido"}$
 - ...

Ejemplo: análisis de signo

- Así podemos abstraer cualquier expresión numérica en una expresión de signo:

$$[[((4 + 8) \times 16) / (3 + 9)]]$$
$$((+ [[+]] +) [[\times]] +) [[/]] (+ [[+]] +)$$
$$(+ [[\times]] +) [[/]] +$$
$$+ [[/]] +$$
$$+$$

Ejemplo: análisis de signo

- Así podemos abstraer cualquier expresión numérica en una expresión de signo:

$[[(4 + 8) \times 16) / (0 \times 9)]]$

$((+ [[+] +) [[x] +) [[/]] (o [[x] +)$

$(+ [[x] +) [[/]] o$

$+ [[/]] o$

“indefinido”

Ejemplo: análisis de signo

- Así podemos abstraer cualquier expresión numérica en una expresión de signo:

$[[(4 + 8) \times 16 / (3 + (-9))]]$




$((+ [[+]] +) [[\times]] +) [[/]] (+ [[+]] -)$

$(+ [[\times]] +) [[/]] \text{“incierto”}$

$+ [[/]] \text{“incierto”}$

“indefinido”

Ejemplo: análisis de signo

- Así podemos abstraer cualquier expresión numérica en una expresión de signo:
 - $[[((4 + 8) \times 16) / (3 + 9)]]$  +
 - $[[((4 + 8) \times 16) / (0 \times 9)]]$  "indefinido"
 - $[[((4 + 8) \times 16) / (3 + (-9))]]$  "indefinido"
- En el último caso, la información es imprecisa
 - porque hacemos una sobre-aproximación.

Ejemplo: análisis de signo

- Aun siendo simple, tiene aplicaciones interesantes:
 - Optimización: las variables + se podrían almacenar como *unsigned int*.
 - Validación de invariantes: se podría verificar valores negativos erróneos (el valor balance de una cuenta, que no sea – ni “incierto”).
 - Corrección: se podría verificar que no existan divisiones por cero.

División por cero

- Combinando el análisis de signo con una ejecución simbólica, puede verificarse la división por cero:

```
z = 24;  
if (x > 0 && y > 0) {  
    w = x * y;  
    z = z / w;  
}
```



```
z = +;  
w = + [[*]] +  
z = + [[/]] +
```

- Como resultado z es +. No hay división por cero.

División por cero

- Combinando el análisis de signo con una ejecución simbólica, puede verificarse la división por cero:

```
z = 24;  
if (x > 0) {  
    w = x * y;  
    z = z / w;  
}
```



```
z = +;  
w = + [[*]] "incierto"  
z = + [[/]] "incierto"
```

- Como resultado z es "indefinido". Puede haber división por cero!

Análisis estático en compiladores

- Chequeo de tipos en operaciones
 - No sumar enteros con booleanos.
- Chequeo de parámetros
 - Cantidad y tipo.
- Variables indefinidas
 - `Object o; o.foo();`

Análisis estático en compiladores

- Optimizaciones de código de máquina

- $x+x$ en lugar de $2*x$

- Optimizaciones de ciclos

```
int a = 7, b = 6, sum, z, i;  
for (i = 0; i < 25; i++) {  
    z = a + b;  
    sum = sum + z + i;  
}
```

- *Inlining* de funciones

Análisis estático para *bugfinding*

- Dereferenciamiento de punteros
- *Buffer overflow*
- *Memory leaks*
- *Race conditions*
- Estilo y buenas prácticas

Análisis estático para la mejora del código

- El análisis estático no es una bala de plata
 - no asegura que el código es de buena calidad o correcto.
- Otras técnicas son valiosas, incluso mas:
 - Diseño cuidadoso
 - Testing
 - Revisión de código

Análisis estático para la mejora del código

- Lo importante es encontrar una buena combinación de técnicas.
- El análisis estático es muy efectivo para encontrar ciertos tipos de problemas
 - pero también requiere esfuerzo (falsos positivos)
- Una vez automatizado dentro del proceso de desarrollo se vuelve mas efectivo y eficiente.

Demo

- Findbugs y PMD

Introducción al análisis estático

