Análisis Automático de Programas

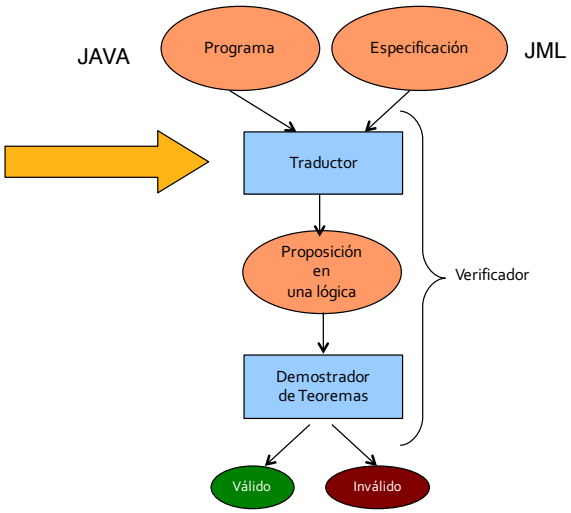
Verificación de programas
usando demostradores

Diego Garbervetsky
Departamento de Computación
FCEyN – UBA

Herramientas para verificar JML

- Diferentes fortalezas
- **Runtime checkers**: Errores reales
  - jmlrac: Encuentra violaciones de aserciones en runtime
  - Jmlunit: Ayuda a la automatización del test de unidad
- **Extender static checkers**: Mejor cobertura
  - ESC/Java2: Encuentra errores y violaciones de contratos
  - Automáticos
- **Buscadores de Modelos**: Buscan contraejemplos
  - JMLForge / JDynAlloy
- **Verificación total**: Garantías
  - Key/Loop/Jack/Jive: Verificación
  - Asistidos por computadora (pero no automáticos)

Verificación de programas



Traduciendo un programa en un teorema

- Tanto el programa como el contrato deben ser traducidos a una lógica en común
- Necesario representar el programa como una fórmula de alguna lógica
- Necesitamos conocer semántica formal del lenguaje
  - Varias enfoques:
    - **Operacional**: Simular ejecución en una máquina "virtual"
    - **Denotacional**: El programa visto como función matemática
    - **Axiomática**: El programa visto como conjuntos de axiomas y reglas de inferencia

Semántica axiomática

- Triplas de Hoare
- Sistema de reglas ideado para la verificación de programas imperativos
- **Correctitud parcial**: {A} `código` {B} si
  - el programa comienza en un estado que cumple **A**
  - Si la ejecución de `código` termina
  - Entonces: el estado final cumple **B**

Semántica axiomática

- Lenguaje imperativo
  - Básicas
    - **Nada:** `skip`
    - **Asignación**: `x:= E;`
  - Control de flujo
    - **Secuencia:** `S1; S2`
    - **Condicional:** `if (cond) {S1} else {S2}`
    - **Iteración:** `while (cond) {S}`

Reglas de Hoare

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\frac{\{A\}\ s1\ \{C\} \quad \{C\}\ s2\ \{B\}}{\{A\}\ \texttt{s1;s2}\ \{B\}}$$

$$\frac{\{A \,\&\&\, cond\}\ s1\ \{B\} \quad \{A \,\&\&\, !cond\}\ s2\ \{B\}}{\{A\}\ \texttt{if(cond) \{s1\} else \{s2\}}\ \{B\}}$$

$$\frac{\{A \,\&\&\, cond\}\ body\ \{A\} \qquad (A \,\&\&\, !cond) \Rightarrow B}{\{A\}\ \texttt{while(cond) \{body\}}\ \{B\}}$$

Reglas de Hoare: asignación

Regla **forward**:

$$\{ A \}\ \mathbf{x:=\ E}\ \{\exists x'.A[x \to x'] \ \&\&\ x == E[x \to x']\}$$

- Intuición: x' es el valor anterior de x. (\old(x))
- Ejemplo:

$$\{ x>=3 \}\ \mathbf{x:=\ x+2}\ \{\exists x'.(x>=3)[x \to x'] \ \&\&\ x == (x+2)[x \to x']\}$$

$$\{ x>=3 \}\ \mathbf{x:=\ x+2}\ \{\exists x'.x'>=3 \ \&\&\ x == x'+2\}$$

$$\{ x>=3 \}\ \mathbf{x:=\ x+2}\ \{\exists x'.x'>=3 \ \&\&\ x-2== x'\}$$

$$\{ x>=3 \}\ \mathbf{x:=\ x+2}\ \{x-2>=3\}$$

$$\{ x>=3 \}\ \mathbf{x:=\ x+2}\ \{x>=5\}$$

Reglas de Hoare: asignación

Regla **backward**:

$$\{ B[x \rightarrow E] \}\ \mathbf{x := E}\ \{B\}$$

- Intuición: Para que **B** valga luego de la asignación, la precondición debe tomar en cuenta la relacion **previa** entre x y E

▪Ejemplo:

{?} **x:=4** {x==4}
{x==4[x→4]} **x:=4** {x==4}
{4==4} **x:=4** {x==4}
{true} **x:=4** {x==4}

▪Otro ejemplo:

{?} **x:= x+2** {x>=5}
{x>=5[x→x+2]} **x:=x+2** {x>=5}
{x+2>=5} **x:=x+2** {x>=5}
{x>=3} **x:=x+2** {x>=5}

Traducción de programas a fórmulas

- **Verification condition** (VC)
  - Fórmula que si es válida (verdadera) significa que el programa cumple con el contrato
- Dado un contrato de M:
  - preM: precondición (requires)
  - posM: poscondición (ensures)
- Básicamente 2 enfoques
  - **Backward**: VC:= preM => **VCb**(cuerpo, posM)
  - **Forward**: VC:= **VCf**(cuerpo, preM) => posM

VC backwards

- Basado en el cálculo de precondiciones más débiles (WP)
- Idea WP(B,s): Dado una poscondición **B** y una instrucción **s** determinar cuál es el predicado menos restrictivo que garantiza que se llegue a **B** por **s**

{WP?}
x := x + 1
{ x>=5 && y<0 }

{ x>= 10 && x< 15}?
x := x + 1
{ x>=5 && y<0 }

{ x>= 10 && x< 15 && y<-10}
x := x + 1
{ x>=5 && y<0 }

{ x>= 10 && x< 15 && **y<0**}
x := x + 1
{ x>=5 && y<0 }

{ x>= 4 && y<0}
x := x + 1
{ x>=5 && y<0 }

Verificando el contrato

- Si tenemos un mecanismo automático y exacto de calcular la WP estamos salvados!
- **VCb**(cuerpo, posM) = WP(cuerpo, posM)
- **Backward** : preM => WP(cuerpo, posM)

```
public m()
modifies x,y;
requires x>10 && x<=15 && y<-1
ensures x>=5 && y<0;
{
    x:=x+1;
}
```

{ x>= 4 && y<0}
x := x + 1
{ x>=5 && y<0 }

x>10 && x<=15 && y<-1
=> x>= 4 && y<0 ✓

Verification condition

Como se calcula la WP

- WP(skip, B) $=_{def}$ B
- WP(x:=E, B) $=_{def}$ B[x→E]
- WP(s1;s2 , B) $=_{def}$ WP(s1, WP(s2, B))
- WP(if(E){s1}else{s2}, B) $=_{def}$

  E=> WP(s1,B) &&

  !E => WP(s2,B)

Ejemplo

- WP(skip, B) $=_{def}$ B
- WP(x:=E, B) $=_{def}$ B[x→E]
- WP(s1;s2 , B) $=_{def}$ WP(s1, WP(s2, B))
- WP(if(E){s1}else{s2}, B) $=_{def}$ E=> WP(s1,B) && !E => WP(s2,B)

```
bool P(bool a, bool b)
requires true
ensures c==a || b
{
  if (a)
    c=true
  else
    c=b
}
```

WP(if(a)..., c==a||b) =

a=> WP(c=true, c==a||b) &&

!a => WP(c=b, c==a||b)

= (a => true==a||b) && (!a => b== a||b)

Conjetura lógica a probar:

preM => WP(P, c==a||b)

true =>(a=> true==a||b) && (!a => b==a||b) ✓

Cual es el problema con la generación de WP?

- **Los ciclos!**
- WP_k(while(E) {S}, B)
  - WP_0(...) $=_{def}$!E => B
  - WP_1(...) $=_{def}$!E => B && E => WP(S,B)

    = WP_0(...) && E => WP(S,B)
  - WP_2(...) $=_{def}$ WP_1(...) && E=>WP(S, WP_1(...))
  - ....
  - WP_i+1(...) $=_{def}$ WP_i && E=>WP(S,WP_i(...))

- Calcularla de forma precisa puede ser imposible en general...

Lidiando con los ciclos

- Soluciones
- Hacer **loop unrolling**: Analizar un conjunto limitado de iteraciones
- Anotar los programas con invariantes de ciclos

Loop unrolling

```
public m(int n)
modifies x,y;
requires x>0 && n>0;
ensures x>=n
{
   for(int i=0; i< n; i++)
   {
        x=x+1;
   }
}
```

```
public m'(int n)
modifies x,y;
requires x>0 && n>0;
ensures x>=n
{
    int i=0;
    if(n>=1) x=x+1; i=1;
    if(n>=2) x=x+1; i=2;
    if(n>=3) x=x+1; i=3;
    assume n<=3;
}
```

- La WP (m, x>=n) real es **x>=0**
- Con 3 unrolls, WP(m' , x>=n) = **(n<=3 => x>=0)**
  - Vale que x>0 && n>0 => (n<=3 => x>=0)
  - Pero unrolling produjo algo más débil…
    - Hint: Que pasa para n = 4 (y otra pre de m')?

Loop unrolling → Unsafe

```
public m(int n)
modifies x;
requires x>0 && n>0;
ensures x>1
{
   for(int i=0; i< n; i++)
   {
        x=x+1;
        if(i>5) x=0;
   }
}
```

```
public m()
modifies x,y;
requires x>0 && n>0;
ensures x>1
{
   int i=0;
   if(n>=1) {
     x=x+1; i=1
   }
   if(n>=2) {
      x=x+1; i=2;
   }
   assume n<=2;
}
```

- Con 2 unrols
  - WP(m' , x>1) = **(2>=n) => x>-1**
- Como x>0 && n>0 => ((2>=n) => x>-1), dice que el programa es correcto!
- Pero si n>6 el programa no cumple con su contrato!
  - La wp correcta es **x>0 && n>0 && n<=6**

Verificación usando invariantes

- Invariante: codifica el "proceso" que realiza el ciclo

```
public m(int n)
modifies x;
requires x>0 && n>0;
ensures x>1
{
   for(int i=0; i< n; i++)
      invariant i>=0 && i<=n && n>0
                &&(i<=6=>x>i)&&(i>6=>x==0)
   {
        x=x+1;
        if(i>5) x = 0;
   }
}
```

Teorema del invariante:
P => Inv
{B && Inv} S {Inv}
(!B && Inv) => Q
{P} while (B) {S} {Q}

P = {i==0 && x>0 && n>0} ; Q={x>1}
I = {i>=0 && i<=n && n>0 &&(i<=6=>x>i) && (i>6=>x==0)}

Verificando con invariantes

Teorema del invariante:
P => Inv
{B && Inv} S {Inv}
(!B && Inv) => Q
{P} while (B) {S} {Q}

P = {i==0 && x>0 && n>0} y Q={x>1}
I = {i>=0 && i<=n && n>0
&&(i<=6=>x>i) && (i>6=>x==0)
}

Antes de entrar. Vale?
(i==0 && x>0 && n>0) =>
i>=0 && i<=n && n>0 &&(i<=6=>x>i)&&(i>6=>x==0)?
Si! (notar i==0)

En cualquier iteración:
i>=0 && i<n && n>0&&(i<=6=>x>i)&&(i>6=>x==0)
{x'=x+1, if(i>5)x'=0, i'=i+1}
i'>=0 && i'<=n && n>0&&(i'<=6=>x'>i')&&(i'>6=>x'==0)?
Si! (caso interesante cuando i==6)

Verificando con invariantes

Teorema del invariante:
P => Inv
{B && Inv} S {Inv}
(!B && Inv) => Q
{P} while (B) {S} {Q}

P = {i==0 && x>0 && n>0} y Q={x>1}
I = {i>=0 && i<=n && n>0
&&(i<=6=>x>i) && (i>6=>x==0)
}

Al salir? i>=n, osea i==n (reemplazamos i por n)
i==n && n>0&&(n<=6=>x>**n**)&&(**n**>6=>x==0)
=>**x>1**?

NO!

- x>n>0 => x>1
requiere n<=6
- x ==0
requiere n>6

- Moraleja:
  - O esta mal la precondición (falta n<=6)
  - O esta mal la poscondición (x==0 || x>1)

Recapitulando

- **VC backward**: preM => WP(codigoM, posM)
  - Sea P = WP(codigoM, posM) exacta
  - En caso de ciclos 2 opciones:
  - Sea, PU = WP_unrolling y PI = WP_invariant
    - PreM implica PU pero PU **no implica** P! (unsafe)
      - PreM => PU**=>?** P
      - PU demasiado débil
    - PI => P (safe!) pero PreM puede no implicar PI (conservador)
      - PreM **=>?** PI => P
      - PI demasiado fuerte

Tratamiento de llamadas

- Opciones:
  - Inlining
  - Usar el contrato

```
//@ requires P;
//@ ensures Q;
//@ modifies M;

public void m() {
  …
}
```

Genera en el método

```
public void m() {
//@ assume P;
…
//@ assert Q;
}
```

Y en el llamador…

```
//@ assert P;
o.m(); (havoc M)
//@ assume Q;
```

Herramientas para verificar JML

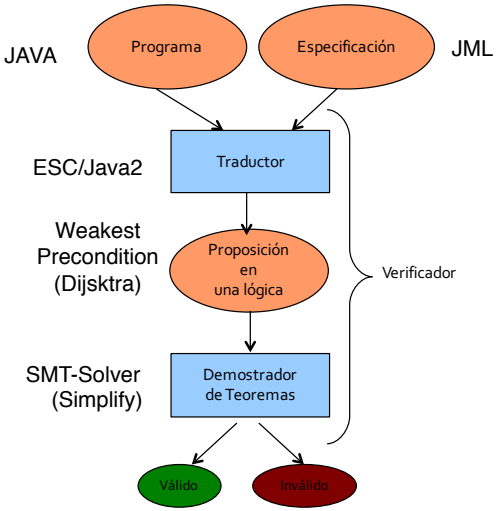
- Diferentes fortalezas
- Runtime checkers: Errores reales
  - jmlrac: Encuentra violaciones de aserciones en runtime
  - Jmlunit: Ayuda a la automatización del test de unidad
- Extended static checkers: Mejor cobertura
  - ESC/Java2: Encuentra errores y violaciones de contratos
  - Automáticos
- Buscadores de Modelos: Buscan contraejemplos
  - JMLForge / JDynAlloy
- Verificación: Garantías: Utilizando
  - Key/Loop/Jack/Jive: Verificación
  - Asistidos por computadora (pero no automáticos)

ESC/Java2 / OpenJML

- ESC: Extended Static Checker
- Desarrollado originalmente en Compaq SRC
  - Compartido como ESC/Java2
- Lenguaje de especificación: JML
- Encontrar bugs / aumentar calidad

Verificación de programas

- Lenguaje de programación
- Lenguaje de especificación
- Representación lógica del programa
- Procedimiento automático de verificación



Algunas características

- Lee casi toda la sintaxis de JML
- Chequea casi toda la semántica de JML
  - Por ejemplo: no chequea \reach()
- Realiza varios chequeos además de los contratos
  - Null, array index out of bounds….
- Permite utilizar diferentes demostradores
- Se "integra" a Eclipse

ESC/Java2: verificador modular (1)

- Razona sobre cada método individualmente:

```
class A {
  byte[] b;

  public void n() { b = new byte[20];}

  public void m() { n(); b[0]=2; … }
}
```

- ESC/Java2 alerta que b[0] puede ser una referencia a null, aunque no existe ejecución alguna del código que lo permita

ESC/Java2: verificador modular (2)

- El razonamiento utiliza sólo los contratos

```
class A {
 byte[] b;
 //@ ensures b!=null && b.length = 20;
 public void n() { b = new byte[20];}

 public void m() { n(); b[0]=2; ... }
}
```

- Para verificar correctamente, hay que hacer explícito el comportamiento requerido/esperado de un método

ESC/Java2: incompleto

- ESC/Java2 alerta sobre warnings en programas complicados

```
/*@ requires 0<n;
 @ ensures \result ==
 @ (\exists int x,y,z;
 @       Math.pow(x,n)+Math.pow(y,n)==Math.pow(z,n));
 @*/
public static boolean m(int n) { return n==2; }
```

- Warning: postcondición posiblemente no satisfecha (El demostrador de teoremas tardó demasiado)

ESC/Java2: unsound

- ESC/Java2 puede fallar en producir un warning en programas incorrectos

```
public class Positive {
 private int n=1;
 //@ invariant n>0;
 public void increase() {
  //n=(int)Math.pow(2,31)-1;
  n++;
 }
}
```

- ESC/Java2 no produce warnings, pero increase puede romper el invariante, si n es (2^31)-1

Ejemplo

```
class Purse {
  int money;
  //@ invariant money >= 0;
}
```

- ¿ El método **loseMoney** es correcto ?
- ¿ Qué comportamiento esperamos de ESC/Java2 ?

```
class RichPerson {

 String mansion_address;
 Purse purse;

 //@ invariant purse.money > 10;

 public void earnMoney() {
   purse.money = purse.money +1;
 }
```

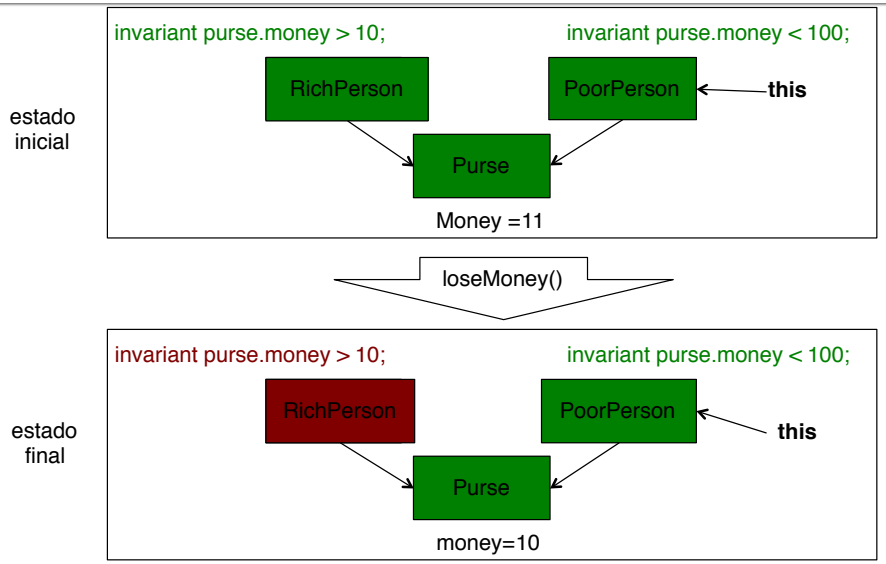
```
class PoorPerson {

 String slum_address;
 Purse purse;

 //@ invariant purse.money < 100;

 //@ requires purse.money > 0;
 public void loseMoney() {
   purse.money = purse.money -1;
 }
```

Posible violación del contrato



Ejemplo

- ESC/Java2 con PoorPerson, RichPerson, Purse
  - `PoorPerson: loseMoney() …`
  - `[0.197 s 39390680 bytes]  passed`
- ESC/Java2 no encontró el bug :
- No analizó si se mantiene o no el invariante de los objetos de la clase RichPerson
  - No lo considero como una clase a analizar

Arreglando el problema…

```
public void loseMoney(RichPerson
my_friend) {
  purse.money = purse.money -1;
}
```

- escj -routine loseMoney *.java
  - `PoorPerson.java:24: Warning: Possible violation of object invariant (Invariant)`
  - `Associated declaration is "RichPerson.java", line 13, col 6:  //@ invariant purse.money > 10;`
- Agregar un argumento de la clase RichPerson forzó la verificación de sus objetos

Fortalezas

- Completamente automática
  - No requiere intervención humana para las demostraciones
- Es robusta (supuestamente)
- Da feedback incluso sin especifaciones
- Intregrada con Eclipse
- Es popular
  - Se usa bastante

Debilidades

- No es completo ni sound
  - Tira falsos positivos y negativos
- A pesar de los esfuerzos requiere bastantes anotaciones
  - Por ejemplo: invariantes de ciclos si queremos ganar soundness
- No está bien documentada
- La respuesta que da al correr no es fácil de descifrar
  - Requiere cierta experiencia

Referencias

- ESC/Java2
  - http://kindsoftware.com/products/opensource/ESCJava2/
  - http://jmlspecs.sourceforge.net/
- Curso ECI 2007:
  - http://www.dc.uba.ar/events/eci/2007/courses/n3
- Dafny: http://dafny.codeplex.com/
- Spec#: http://research.microsoft.com/en-us/projects/specsharp/