Bases Formales de la Computación

Gerardo M. Sarria M.

Pontificia Universidad Javeriana

22 de agosto de 2008

Gerardo M.
Sarria M.

VERIFICACIÓN

Gerardo M.
Sarria M.

Contenido

Sistemas

Los Sistemas de Software se han convertido en una pieza clave de nuestras vidas:

- Sistemas de transacción bancarias
- Aplicaciones comerciales en internet
- Tarjetas inteligentes

Los Sistemas de Software son más importantes ahora ya que han penetrado en nuestro entorno y tanto los proveedores como los clientes de dichos sistemas se están comenzando a preocupar por su correctitud.

Sistemas

Sin embargo, dado que los Sistemas de Software . . .

- son cada vez más complejos,
- tienen comportamientos disímiles,
- admiten múltiples puntos de vista

Sistemas

Sin embargo, dado que los Sistemas de Software . . .

- son cada vez más complejos,
- tienen comportamientos disímiles,
- admiten múltiples puntos de vista

son propensos a fallar

Sistemas

Las fallas son inaceptables en los sistemas de software actuales.

No es bueno apagar un sistema que no funcione bien para volver a un estado seguro

Hay que pensar en los sistemas críticos en donde hay vidas humanas envueltas

Sistemas

Errores en sistemas de hardware y software que han sido muy costosos:

- El *bug* aritmético de punto flotante del Intel Pentium costó $\$500M$ (1994)
- El cohete Ariane 5 perdió $\$7B$ (1996)
- El choque del Mars Polar Lander costó $\$120M$ (2000)

Verificación

La verificación es usada en diferentes contextos. Como . . .

- un proceso para obtener la prueba de correctitud formal de un sistema por medio de deducciones (theorem proving)
- cualquier acción tomada para encontrar errores en un programa usando verificación automática (model-checking)

Verificación

La verificación es usada en diferentes contextos. Como . . .

- un proceso para obtener la prueba de correctitud formal de un sistema por medio de deducciones (theorem proving)
- cualquier acción tomada para encontrar errores en un programa usando verificación automática (model-checking)

La Etapa de Pruebas en ingeniería de software:

- No es un un técnica de verificación
- Es más cercana a un muestreo que a una prueba de correctitud exhaustiva

Limitaciones

- Los métodos de verificación no garantizan la correctitud del código actual. Ellos verifican un modelo abstracto.

Limitaciones

- Los métodos de verificación no garantizan la correctitud del código actual. Ellos verifican un modelo abstracto.
- Las pruebas de correctitud pueden ser de por sí incorrectas

Limitaciones

- Los métodos de verificación no garantizan la correctitud del código actual. Ellos verifican un modelo abstracto.
- Las pruebas de correctitud pueden ser de por sí incorrectas
- El proceso de verificación solo captura un pequeña parte de la funcionalidad de un sistema

Limitaciones

- Los métodos de verificación no garantizan la correctitud del código actual. Ellos verifican un modelo abstracto.
- Las pruebas de correctitud pueden ser de por sí incorrectas
- El proceso de verificación solo captura un pequeña parte de la funcionalidad de un sistema
- La verificación se hace con respecto a una especificación dada, la cual se forma de manera manual, y puede estar incompleta

Limitaciones

- Los métodos de verificación no garantizan la correctitud del código actual. Ellos verifican un modelo abstracto.
- Las pruebas de correctitud pueden ser de por sí incorrectas
- El proceso de verificación solo captura un pequeña parte de la funcionalidad de un sistema
- La verificación se hace con respecto a una especificación dada, la cual se forma de manera manual, y puede estar incompleta
- Las técnicas de verificación automática (model-checking) están restrigidas a sistemas de estado finito

Prejuicios

- Los métodos formales solo pueden ser usados por matemáticos
- El uso de métodos formales harán lentos los proyectos
- No es visual (faltan flechas y cajas)
- El proceso de verificación de por sí propenso a errores, así que ¿por qué molestarse con ellos?

Dificultades Reales

- Se debe pensar mucho antes de programar
- Cómo incorporar esto en el proceso de desarrollo
- La construcción de modelos no es fácil
- Tecnología de pruebas automáticas debe mejorar
- Mala calidad de los documentos de requerimientos

Aplicación

Modelado Convertir un diseño en un formalismo. Se puede necesitar abstraer elementos irrelevantes o detalles sin importancia.

Aplicación

Modelado Convertir un diseño en un formalismo. Se puede necesitar abstraer elementos irrelevantes o detalles sin importancia.

Especificación Indicar las propiedades que debe satisfacer el diseño. La especificación es dada usando un formalismo lógico (lógica temporal). Es necesario que la especificación esté completa, es decir, cubra todas las propiedades que el sistema debe satisfacer.

Aplicación

Modelado Convertir un diseño en un formalismo. Se puede necesitar abstraer elementos irrelevantes o detalles sin importancia.

Especificación Indicar las propiedades que debe satisfacer el diseño. La especificación es dada usando un formalismo lógico (lógica temporal). Es necesario que la especificación esté completa, es decir, cubra todas las propiedades que el sistema debe satisfacer.

Verificación Ideal sería que fuera completamente automática. En la práctica necesita asistencia humana: análisis de los resultados.

Sistemas

Los Sistemas de Software son modelos de sistemas reales que tienen un comportamiento coherente.

Cada característica que se elige observar de un sistema real se representa mediante una variable.

El valor de estas variables para todos los objetos del sistema en un momento dado es una observación.

Observaciones

Ejemplo:
Suponga que un sistema consta de tres objetos.
De ellos se observa una característica que se representa por una variable p_i.

Una observación podría ser:

$$p_1 = 3 \ \wedge \ p_2 = 9 \ \wedge \ p_3 = 2$$

Observaciones

Ejemplo:
Suponga que un sistema consta de tres objetos.
De ellos se observa una característica que se representa por una variable p_i.

Una observación podría ser:

$$p_1 = 3 \ \wedge \ p_2 = 9 \ \wedge \ p_3 = 2$$

Definir los objetos y las características observables corresponde a la parte estática de la descripción de un sistema.

Observaciones

El número de observaciones de un sistema puede ser inmenso (o infinito!).

Ejemplo:
Suponga que en el mismo sistema se ha decidido observar características representadas por una variable z y una variable y.
Se recolectan las siguientes observaciones:

$$z = 2 \wedge y = 0{,}707$$
$$z = 7 \wedge y = 0{,}378$$
$$z = 9 \wedge y = 0{,}33$$
$$z = 10 \wedge y = 0{,}316$$

Observaciones

Los datos anteriores no permiten formarse una idea del funcionamiento del sistema.

Observaciones

Los datos anteriores no permiten formarse una idea del funcionamiento del sistema.

Solución: Una propiedad o invariante del sistema

Observaciones

Los datos anteriores no permiten formarse una idea del funcionamiento del sistema.

Solución: Una propiedad o invariante del sistema

Para el caso anterior:

$$\sqrt{z} \times y = 1$$

Dinámica de los Sistemas

Pero un sistema NO es, en general, estático.

¿Cómo evoluciona un sistema?

Dinámica de los Sistemas

Pero un sistema NO es, en general, estático.

¿Cómo evoluciona un sistema?

¿Qué ocasiona un cambio en las observaciones?

Dinámica de los Sistemas

Pero un sistema NO es, en general, estático.

¿Cómo evoluciona un sistema?

¿Qué ocasiona un cambio en las observaciones?

EVENTOS

Eventos

Ejemplo:
Un evento puede ocasionar un incremento en el valor de la variable x del sistema anterior.

Ese mismo evento, entonces, tiene que ocasionar una modificación del valor de y, de modo que el invariante se siga cumpliendo.

Eventos

Ejemplo:
Un evento puede ocasionar un incremento en el valor de la variable x del sistema anterior.

Ese mismo evento, entonces, tiene que ocasionar una modificación del valor de y, de modo que el invariante se siga cumpliendo.

Incrementar:
begin
$z, y := z + 1,\ 1/\sqrt{z+1}$
end

Descripción Completa de un Sistema

Parte Estática:

- Variables (características observables)
- Invariante

Parte Dinámica:

- Conjunto de Eventos

B es una métodología para especificar, diseñar y codificar Sistemas.

La descripción del sistema se hace de esta manera:

- El invariante es un predicado
- Los eventos son de la forma:

GUARDA
ACCIÓN

Eventos

Los eventos podrán ser de la siguiente forma básica:

nombre $=$
ANY $x, y, z, \ldots$ **WHERE**
$P(x, y, \ldots, v, w, \ldots)$
THEN
$S(x, y, \ldots, v, w, \ldots)$
END

Eventos

La operación del evento anterior correspondería a los siguientes pasos:

Eventos

La operación del evento anterior correspondería a los siguientes pasos:

1. Escoja valores para las variables locales $x, y, \ldots$ de manera que para esos valores escogidos y para los valores acutales de las variables del sistema, $P(x, y, \ldots, v, w, \ldots)$ se cumpla.

Eventos

La operación del evento anterior correspondería a los siguientes pasos:

1. Escoja valores para las variables locales $x, y, \ldots$ de manera que para esos valores escogidos y para los valores acutales de las variables del sistema, $P(x, y, \ldots, v, w, \ldots)$ se cumpla.
2. Si no fue posible escoger dichos valores, el evento no se activa

Eventos

La operación del evento anterior correspondería a los siguientes pasos:

1. Escoja valores para las variables locales $x, y, \ldots$ de manera que para esos valores escogidos y para los valores acutales de las variables del sistema, $P(x, y, \ldots, v, w, \ldots)$ se cumpla.
2. Si no fue posible escoger dichos valores, el evento no se activa
3. Si la escogencia fue posible, ejecute la acción $S(x, y, \ldots, v, w, \ldots)$

Rodin

Rodin es una herramienta para el desarrollo riguroso de sistemas de software complejos.

La descripción de un sistema se hace de la siguiente manera:

1. Se describe un contexto
2. Se describe una máquina abstracta

Ejemplo: Sistema MIO

- Hay una cierta cantidad de buses
- Hay una cierta cantidad de estaciones
- En un momento dado, en una estación puede estar un bus
- Se observa la llegada y salidad de buses a las estaciones

MIO: Componentes Estáticos - Contexto

- Constante n: Número total de buses
- Constante m: Número total de estaciones

Para estas constantes se definen axiomas que establecen sus propiedades:

variable	**tipo**
n	$n \in \mathcal{N}_1$
m	$m \in \mathcal{N}_1$

Constantes, Conjuntos y sus Propiedades forman el Contexto del sistema

MIO: Componentes Estáticos - Contexto

Gerardo M.
Sarria M.

CONTEXT *buses_estaciones*
CONSTANTS
n
m
AXIOMS
$axm1 : m : \mathbb{N}_1$
$axm2 : n : \mathbb{N}_1$
END

MIO: Componentes Estáticos - Abstracción

- Variable *be*: Número de buses en estaciones
- Propiedades (tipo) de la variable:

$$be \in \mathbb{N}$$

- Observaciones de *be* obedecen a un INVARIANTE

MIO: Componentes Estáticos - Abstracción

- Variable *be*: Número de buses en estaciones
- Propiedades (tipo) de la variable:

$$be \in \mathbb{N}$$

- Observaciones de *be* obedecen a un INVARIANTE

$$be \leq n \ \wedge \ be \leq m$$

MIO: Componentes Dinámicos

Gerardo M. Sarria M.

Inicialización:

$$be := 0$$

MIO: Componentes Dinámicos

Inicialización:

$$be := 0$$

Entra un bus a una estación:

$$\text{where } be < n \ \wedge \ be < m \text{ then } be := be + 1 \text{ end}$$

MIO: Componentes Dinámicos

Inicialización:

$$be := 0$$

Entra un bus a una estación:

$$\text{where } be < n \ \wedge \ be < m \text{ then } be := be + 1 \text{ end}$$

Sale un bus de una estación:

$$\text{where } be > 0 \text{ then } be := be - 1 \text{ end}$$

MIO: Especificación

MACHINE *mio*
SEES *buses_estaciones*
VARIABLES
be
INVARIANT
$inv1 : be : \mathbb{N}$
$inv2 : be \leq n$
$inv3 : be \leq m$
...

MIO: Especificación

EVENTS

initialisation

then

$act1 : be := 0$

end

...

Gerardo M.
Sarria M.

MIO: Especificación

llega
where
$grd1 : be < n$
$grd2 : be < m$
then
$act1 : be := be + 1$
end
sale
where
$grd1 : be > 0$
then
$act1 : be := be - 1$
end
END

Consistencia

El invariante representa los valores válidos de las variables del sistema (estado).

Las acciones de los eventos modifican el estado.

El nuevo estado debe satisfacer el invariante.

Consistencia

Para un modelo con una variable v e invariante $I(v)$, y un evento de la forma:

ANY x **WHERE**
$P(x, v)$
THEN
$v := E(x, v)$
END

la sentencia a probar es

$$I(v) \ \wedge \ P(x, v) \Rightarrow I(E(x, v))$$

Consistencia

Para el caso del MIO, se tiene

$$P(m,n) = m \in \mathbb{N}_1 \ \wedge \ n \in \mathbb{N}_1$$

$$I(m,n,be) = be \in \mathbb{N} \ \wedge \ be \leq n \ \wedge \ be \leq m$$

Consistencia

Para el caso del MIO, se tiene

$$P(m,n) = m \in \mathbb{N}_1 \ \wedge \ n \in \mathbb{N}_1$$
$$I(m,n,be) = be \in \mathbb{N} \ \wedge \ be \leq n \ \wedge \ be \leq m$$

- Factibilidad de la inicialización:

$$m \in \mathbb{N}_1 \ \wedge \ n \in \mathbb{N}_1 \Rightarrow \exists be.be = 0$$

- Inicialización satisface el invariante:

$$m \in \mathbb{N}_1 \ \wedge \ n \in \mathbb{N}_1 \Rightarrow 0 \in \mathbb{N} \ \wedge \ 0 \leq n \ \wedge \ 0 \leq m$$

Ejemplo: Partido de Fútbol

- Se observan personas. Unas están adentro y otras afuera del campo
- Una persona de afuera puede entrar al campo (comienzo del juego)
- Una persona del campo puede cambiarse por una de afuera
- Una persona de adentro puede salir (expulsión)

Partido de Fútbol - Contexto

CONTEXT *partido*
SETS
PERSONAS
AXIOMS
$axm1 : finite(PERSONAS)$
END

Gerardo M.
Sarria M.

Partido de Fútbol - Variables e Invariante

VARIABLES $adentro, afuera$

INVARIANT

$inv1 : adentro \subseteq PERSONAS$

$inv2 : afuera \subseteq PERSONAS$

$inv1 : adentro \cap afuera = \varnothing$

$inv1 : adentro \cup afuera = PERSONAS$

...

Partido de Fútbol - Eventos

EVENTS

initialisation

then

$act1 : adentro := \emptyset$

$act2 : afuera := PERSONAS$

end

...

Partido de Fútbol - Eventos

entra

any j

where

$grd1 : afuera \neq \emptyset$

$grd2 : j \in afuera$

then

$act1 : afuera := afuera \setminus \{j\}$

$act2 : adentro := adentro \cup \{j\}$

end

...

Partido de Fútbol - Eventos

expulsado

any j

where

$grd1 : adentro \neq \emptyset$

$grd2 : j \in adentro$

then

$act1 : adentro := adentro \setminus \{j\}$

$act2 : afuera := afuera \cup \{j\}$

end

...

Partido de Fútbol - Eventos

cambio

any e, s

where

$grd1 : adentro \neq \emptyset$

$grd2 : afuera \neq \emptyset$

$grd3 : s \in adentro$

$grd4 : e \in afuera$

then

$act1 : adentro := (adentro \backslash \{s\}) \cup \{e\}$

$act2 : afuera := (afuera \backslash \{e\}) \cup \{s\}$

end

END

Partido de Fútbol

Ejercicio:

Considere otra especificación del partido de fútbol, que distingue, entre las personas que están adentro, a los jueces.

- Obviamente a quien expulsan no puede ser un juez
- Tampoco se puede cambiar un jugador por un juez

Especifique este sistema

JML

JML es un lenguaje de especificación formal para programas escritos en Java.

La idea es:

- Tener un registro de las desiciones de diseño e implementación
- Especificar el comportamiento de las clases

Las especificaciones JML describen:

- interfaces (nombres e información estática)
- comportamiento (cómo las clases e interfaces actúan cuando son usadas)

JML

Las especificaciones JML son:

- Predicados en lógica de primer orden
- Expresiones booleanas de Java

Las especificaciones se adicionan entre `/*@ ... @*/`, o después de `//@` en código Java

- precondiciones
- postcondiciones
- invariantes

Gerardo M.
Sarria M.

Operadores

- Operadores lógicos de Java (||, &&, !)
- Operadores de relación y corrimiento de Java (<, <=, . . . , <<, >>)
- Implicaciones lógicas (==>, <==)
- Equivalencias lógicas (<==>, <=! =>)
- Cuantificación universal y existencial (`\forall`, `\exists`)

Predicados

Los predicados de JML se expresan en lógica de primer orden, construidos usando

- Booleanos de Java,
- otros operadores (`\old`, `\result`, `\forall`, `\exists`, `\max`, etc.),
- y algunas palabras clave (`requires`, `ensures`, `invariant`, etc.).

Predicados

Ejemplo:

```
public class IntMathOps {
  /*@ public normal_behavior
    @ requires y >= 0;
    @ assignable \nothing;
    @ ensures 0 <= \result &&
    @         \result * \result <= y &&
    @         y < (\result + 1) * (\result + 1);
    @*/
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(u);
  }
}
```

Predicados

- `requires`. Se usa para especificar precondiciones.
- `ensures`. Se usa para especificar postcondiciones.
- `normal_behavior`. El método termina normalmente, sin lanzar una excepción.
- `exceptional_behavior`. El método lanza una excepción.
- `behavior`. El método podría terminar normal o abruptamente.

Especificaciones Livianas

- No usan el predicado behavior
- Se escriben en lineas individuales después de //@

```
public class IntMathOps {
  //@ requires y >= 0;
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(y);
  }
}
```

Especificaciones Livianas

- Si una palabra clave `requires` es olvidada, una especificación `requires true;` se asume
- Si una palabra clave `assignable` es olvidada, una especificación `assignable \everything;` se asume
- Si una palabra clave `ensures` es olvidada, una especificación `ensures true;` se asume

JML y Diseño por Contrato

Las precondiciones y postcondiciones de los métodos definen un contrato entre el método (la clase) y el objeto que llama el método.

Este contrato estipula que

1. Los métodos pueden asumir precondiciones y tienen que asegurar postcondiciones
2. Los clientes tienen que asegurar precondiciones y pueden asumir poscondiciones

Gerardo M.
Sarria M.

Contratos y Pruebas Obligatorias

```
class Decimal {
  int intPart, decPart;
  //@ invariant decPart >= 0;

  /*@ requires m != null;
    @ ensures decPart == m.decPart &&
    @         intPart == \old(intPart);
    @*/
  void setDecimal(Decimal m) {
    decPart = m.decPart;
  }
}
```

Contratos y Pruebas Obligatorias

Gerardo M. Sarria M.

Para el método `setDecimal`:

- Si `m != null` y `m.decPart >= 0` entonces
  - `decPart == m.decPart`, y
  - `intPart == \old(intPart)`

Contratos y Pruebas Obligatorias

Para el método `setDecimal`:

- Si `m != null` y `m.decPart >= 0` entonces
  - `decPart == m.decPart`, y
  - `intPart == \old(intPart)`

Para el objeto que hace el llamado:

- `o != null` y `o.decPart >= 0` tiene que ser cierto en todo sitio donde se haga el llamado al método `setDecimal(o)`
- La postcondición de `setDecimal` debe ser asumida en aquellos sitios

Contratos y Pruebas Obligatorias

Para el método `setDecimal`:

- Si `m != null` y `m.decPart >= 0` entonces
  - `decPart == m.decPart`, y
  - `intPart == \old(intPart)`

Para el objeto que hace el llamado:

- `o != null` y `o.decPart >= 0` tiene que ser cierto en todo sitio donde se haga el llamado al método `setDecimal(o)`
- La postcondición de `setDecimal` debe ser asumida en aquellos sitios

Adicionalmente:

- `decPart >= 0` debe ser una invariante de la clase `Decimal`

Ejemplo: Clase Decimal

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
  private short intPart = (short) 0;
  private short decPart = (short) 0;

  public Decimal setValue(Decimal d) throws DecimalException {
    return setValue(d.getIntPart(),d.getDecPart());
  }

  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) decimal_exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
  ...
}
```

\forall y \exists

```
(\forall int i; a[i] != null);

(\forall int i; (0 <= i && i < length) ==>
                src[srcOff + i] == dest[destOff + i]);

(\exists int i; (0 <= i && < length) &&
     (\forall int j; (0 <= j && j < i) ==>
          src[srcOff + j] == dest[destOff + j]));
```

assert

`assert P`:

- `P` tiene que ser cierto en un cierto punto en el cuerpo de un método
- `P` es un propiedad JML válida en la lógica

assert

`assert P`:

- `P` tiene que ser cierto en un cierto punto en el cuerpo de un método
- `P` es un propiedad JML válida en la lógica

```
if (i <= 0 || j < 0) {
   ...
}
else if (j < 5) {
   //@ assert i > 0 && 0 < j && j < 5;
   ...
}
else {
   //@ assert i > 0 && j > 5;
   ...
}
```

assert

JML no permite campos `private` en especificaciones `public`

Poner `/*@ spec_public @*/` antes de una declaración de campo `private` causa que el campo sea incluido en el alcance de toda especificación

assert

JML no permite campos `private` en especificaciones `public`

Poner `/*@ spec_public @*/` antes de una declaración de campo `private` causa que el campo sea incluido en el alcance de toda especificación

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
  /*@ spec_public @*/ private short intPart = (short) 0;
  /*@ spec_public @*/ private short decPart = (short) 0;
  ...
}
```

requires, ensures y normal_behavior

- `requires R`
  Precondición `R`
- `ensures Q`
  Postcondición `Q`
  `Q` tiene que ser cierto si el método termina normalmente (i.e. sin lanzar una `java.lang.Exception`)
- `normal_behavior`(total correctness)
  Si la precondición es cierta el estado previo, entonces el método termina en un estado normal, y la postcondición es cierta en este estado

requires, ensures y normal_behavior

Gerardo M.
Sarria M.


```
public class Decimal extends Object {
  /*@ normal_behavior;
    @ requires i>=0 && d>=0;
    @ ensures intPart == i && decPart == d;
    @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
}
```

signals y exceptional_behavior

- `signals (E e) R`
  Postcondición Excepcional `R`
  `R` tiene que ser cierto si el método lanza una excepción `e` que es subclase de `E`
- `exceptional_behavior`(total correctness)
  Si la precondición es cierta el estado previo, entonces el método terminará en un estado excepcional lanzando una `java.lang.Exception`, y la postcondición excepcional es cierta en este estado

requires, ensures y normal_behavior

Gerardo M.
Sarria M.


```
public class Decimal extends Object {
  /*@ exceptional_behavior;
    @ requires i>=0 && d>=0;
    @ signals (DecimalException e) true;
    @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
}
```

assignable

`assignable L`:

- Un método solo puede modificar el conjunto de ubicaciones de memoria denotadas por `L`
- Cualquier otra ubicación no listada puede entonces no ser modificada
- esto es cierto para postcondiciones normal y excepcional

assignable

assignable L:

- Un método solo puede modificar el conjunto de ubicaciones de memoria denotadas por L
- Cualquier otra ubicación no listada puede entonces no ser modificada
- esto es cierto para postcondiciones normal y excepcional

```
public class Decimal extends Object {
  /*@ normal_behavior;
    @ requires i>=0 && d>=0;
    @ assignable intPart, decPart;
    @ ensures true;
    @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
}
```

Gerardo M.
Sarria M.

\old y \fresh

- `\old(e)`
  Valor de una expresión `r` en el estado previo
- `\fresh(x)`
  `x` no es nulo y no fue declarado en el estado previo

\old y \fresh

- \old(e)
  Valor de una expresión r en el estado previo
- \fresh(x)
  x no es nulo y no fue declarado en el estado previo

```
/*@ behavior
  @ requires true;
  @ assignable decimal;
  @ ensures decimal == i * PRECISION + d;
  @ signals (DecimalException e) (i<0 || d<0) &&
  @                              decimal == \old(decimal) &&
  @                              \fresh(e);
  @*/
public Decimal setValue(short i, short d) throws DecimalException {
  ...
}
```

\result

Representa el valor retornado por un método y tiene el mismo tipo que el método

\result

Gerardo M.
Sarria M.

Representa el valor retornado por un método y tiene el mismo tipo que el método

```
/*@ normal behavior
  @ requires true;
  @ ensures \result <==>
  @             (\exists int i; i>=0 && i<MAX_DATA && data[i]==cur);
  @*/
boolean contens(byte cur) {
  boolean resu = false;
  byte i = (byte)0;
  boolean found = false;
  while(i < MAX_DATA && ! resu) {
    if(data[i] == cur) resu = true;
    else i++;
  }
  return resu;
}
```

invariante

Tiene que ser cierto en todos los estados visibles:

- El comienzo y final de cualquier invocación a un método
- El final de la invocación a un constructor

Se asume de manera implícita en el comienzo de una declaración de método, y no tiene que ser cierto durante la ejecuón de un método (i.e. puede ser temporalmente falso)

invariante

Tiene que ser cierto en todos los estados visibles:

- El comienzo y final de cualquier invocación a un método
- El final de la invocación a un constructor

Se asume de manera implícita en el comienzo de una declaración de método, y no tiene que ser cierto durante la ejecuón de un método (i.e. puede ser temporalmente falso)

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
  public short intPart = (short) 0;
  public short decPart = (short) 0;

  /*@ invariant 0 <= intPart && intPart <= MAX_DECIMAL_NUMBER &&
    @           0 <= decPart && decPart < PRECISION;
    @*/
  ...
}
```

non_null

Reemplaza un especificación de invariante sobre un objeto no nulo

```
public class SalerID extends Object implements PartnerID {
  ...

  //@ invariant data != null;
  public byte[] data = new byte[ID_LENGTH];

  ...
}
```

non_null

Reemplaza un especificación de invariante sobre un objeto no nulo

```
public class SalerID extends Object implements PartnerID {
  ...

  public /*@ non null @*/ byte[] data = new byte[ID_LENGTH];

  ...
}
```

Gerardo M.
Sarria M.

Fin de la Presentación