Validación y Verificación de Software

Generación Automática de Tests Unitarios: Generación basada en Constraint Solving (SMT Solving)

PEX: Generación de Casos de Tests para .NET

Genera tests de unidad para programas .NET

Se integra a MS Visual Studio

Apunta a generar test suites pequeñas, con gran cobertura de código

En general, logra buen cubrimiento de código en poco tiempo

Utiliza constraint solving para lograr cobertura de decisión

Generación de casos de tests: un ejemplo

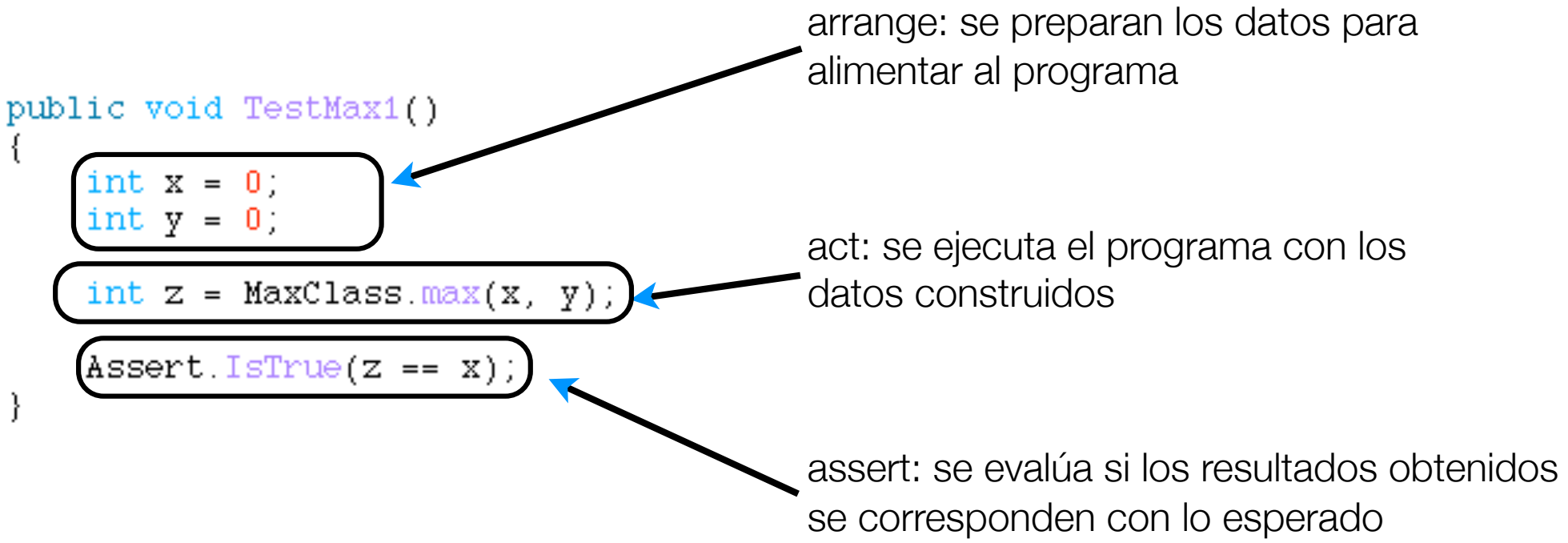
Consideremos el siguiente programa C#:

```
public static int max(int x, int y)
{
    if (x >= y) return x;
    else return y;
}
```

Este es un programa muy simple, que calcula el máximo entre dos valores enteros.

Generación de casos de tests: un ejemplo

Podemos crear manualmente un caso de test para este programa



Generación de casos de tests: un ejemplo

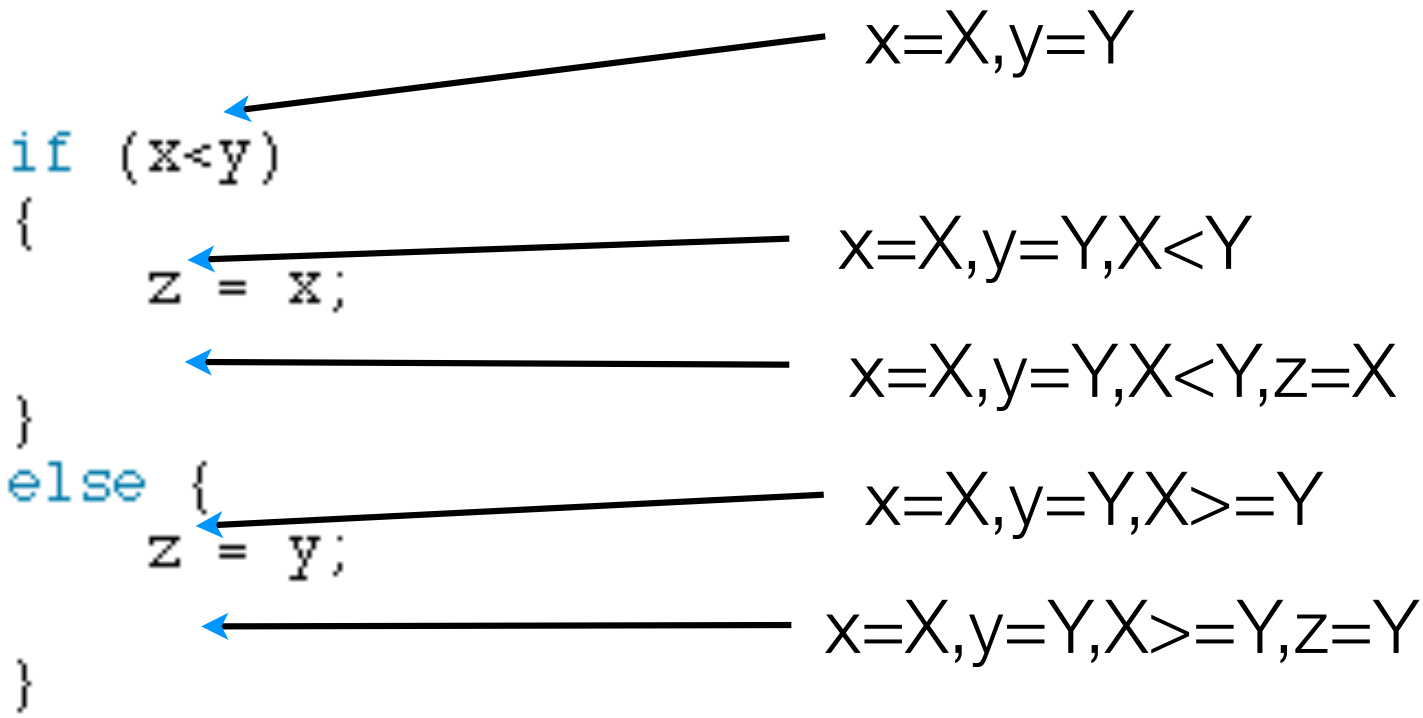
Pex reemplaza el trabajo que hicimos manualmente:

- genera inputs para conseguir cobertura de decisión
  - utilizando técnicas de constraint solving
- genera “asserts” triviales automáticamente (e.g., no se hace null el objeto al cual se aplica una rutina, no se lanzan excepciones, ...)
- permite “guardar” el comportamiento observado en el programa analizado, para contrastar con futuras versiones (test de regresión)

Cómo funciona Pex

Execución Simbólica (Dinámica)

Pex recorre el texto del programa y construye condiciones de caminos:



Se evalúan estas condiciones para conseguir valores que ejerciten las diferentes ramas de ejecución del programa

Ejecución Simbólica Dinámica

se generan entradas aleatorias

Se ejecuta el programa con las entradas concretas y simbólicamente de manera simultánea, guardando condiciones simbólicas en los puntos de decisión

Cuando la ejecución termina, se niega una condición en un punto de decisión para que la próxima ejecución tome otro camino distinto al anterior.

Se invoca un constraint solver con las condiciones de camino obtenidas para generar nuevas entradas que tomen este camino

La ejecución concreta sirve para asistir al constraint solver en la generación de entradas, y verificar que los caminos deseados efectivamente se toman.

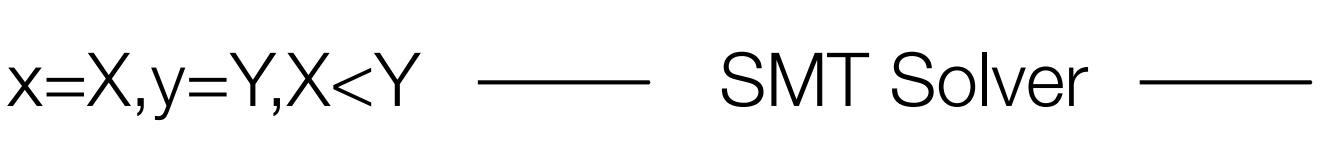
Se repite el procedimiento hasta que todos los caminos deseados se cubren.

Cómo funciona Pex

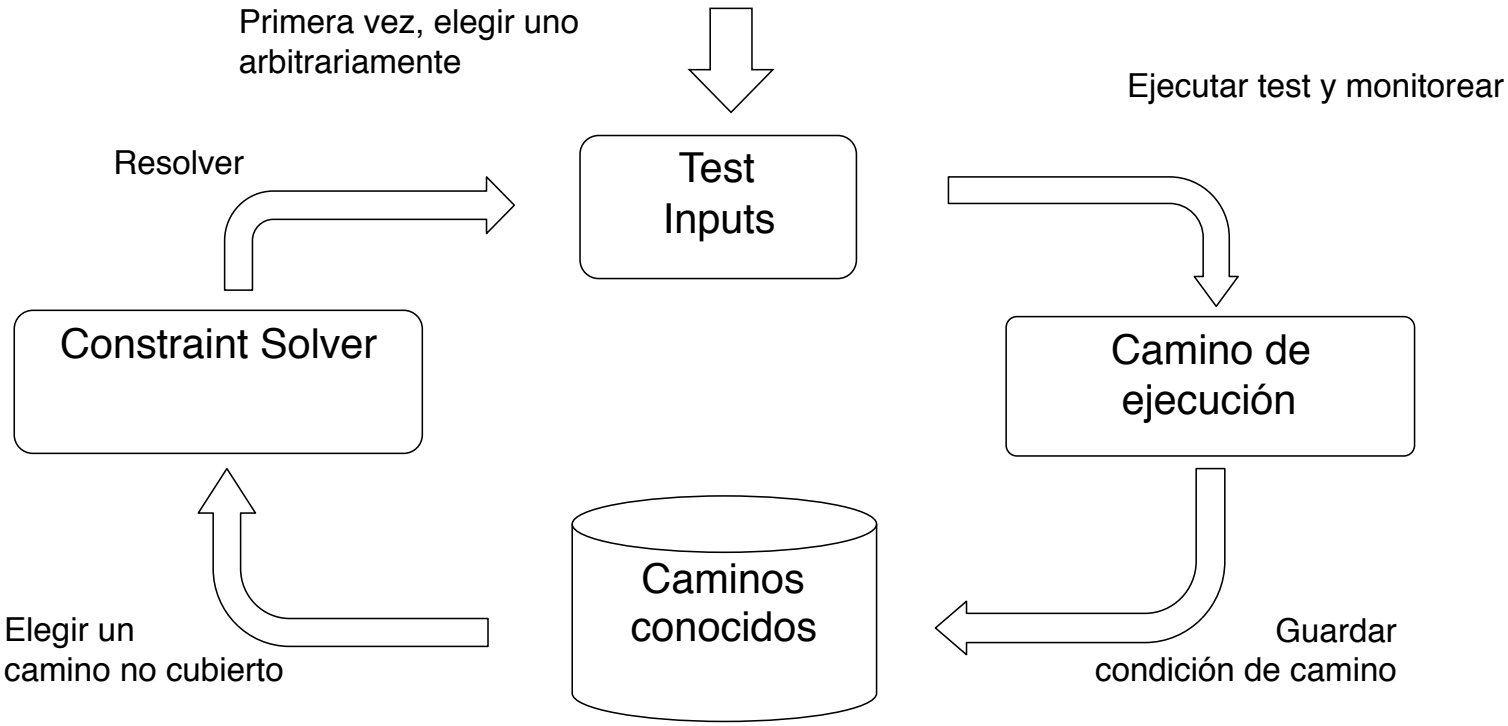
Constraint Solving

Para resolver las restricciones, Pex utiliza SMT solving.

Un SMT solver es similar a un SAT solver: decide si una expresión puede hacerse verdadera, pero soporta tipos más ricos que booleanos, como enteros, reales, algunas estructuras de datos (arreglos), etc.



Cómo funciona Pex



Resultado: test suites pequeñas, alta cobertura

Cuán poderoso es el constraint solver de Pex?

Veamos algunos ejemplos más interesantes

```
public static IComparable maxComp(IComparable x, IComparable y)
{

    if (x.CompareTo(y) >= 0) return x;
    else return y;

}
```

Pex **no** prueba "exhaustivamente" con todas las clases que implementan IComparable: intenta con algunas clases (y advierte al respecto)

Los ciclos se "despliegan" como condicionales anidados: si los valores demandan muchas iteraciones, el tiempo de solving, el número de ramas, la cantidad de constraints pueden ser excesivamente grandes

```
static int mcd_Euclides(int x, int y)
{

    if (x <= 0 || y <= 0) throw new ArgumentException();
    int a = x;
    int b = y;
    while (b != 0)
    {

        if (a > b)
            a = a - b;

        else
            b = b - a;

    }
    return a;
}
```

Tests de Unidad Parametrizados

Simplemente: tests de unidad con parámetros

Pex los genera a partir del código a analizar

es sobre lo que efectivamente se realiza el constraint solving

4A: Assume, Arrange, Act, Assert

Posibilitan incorporar aserciones assume/assert a los casos de tests:

para restringir entradas

para especificar las entradas permitidas

para especificar propiedades esperadas de las salidas

Aserciones vs. Suposiciones

Tanto las aserciones como las suposiciones nos permiten describir propiedades de programas, pero de manera diferente:

Aserción (assert): describe una propiedad que debería satisfacerse en un estado dado

su violación constituye un error

Suposición (assume): describe una propiedad que se supone verdadera en un estado dado

su verdad es supuesta

su violación hace que se descarte la ejecución

si no es verdadera, no es de interés

Tests de Unidad Parametrizados

Restringiendo las entradas válidas

Veamos nuevamente algunos casos anteriores:

```
public static IComparable maxComp(IComparable x, IComparable y)
{

    if (x.CompareTo(y) >= 0) return x;
    else return y;

}
```

Test de unidad parametrizado básico

Se pueden incorporar restricciones a las entradas aquí

```
[PexMethod]
public IComparable maxComp(IComparable x, IComparable y)
{
    PexAssume.IsNotNull(x);
    PexAssume.IsNotNull(y);
    PexAssume.IsTrue(x.GetType()==y.GetType());
    IComparable result = BasicExample.maxComp(x, y);
    return result;
}
```

Tests de Unidad Parametrizados

Restringiendo las entradas para “ayudar a Pex”

Veamos otro de los casos anteriores:

```
static int mcd_Euclides(int x, int y)
{

    if (x <= 0 || y <= 0) throw new ArgumentException();
    int a = x;
    int b = y;
    while (b != 0)
    {

        if (a > b)
            a = a - b;

        else
            b = b - a;

    }
    return a;
}
```

Ahora restringimos las entradas en el test de unidad parametrizado, para evitar que Pex elija valores que hagan “explotar” caminos, condiciones, ramas, ...

```
[PexMethod]
public int mcdEuclides(int x, int y)
{
    PexAssume.IsTrue(x<=10);
    PexAssume.IsTrue(y<=10);
    int result = BasicExample.mcdEuclides(x, y);
    return result;
}
```

Tests de Unidad Parametrizados

Especificando la salida esperada

De manera similar a lo mostrado anteriormente, podemos utilizar aserciones para especificar qué es lo que esperamos del comportamiento del programa analizado.

los “asserts” generados automáticamente por Pex son bastante triviales

```
public void swap()
{
    x = x + y;
    y = x - y;
    x = x - y;
}
```

La sección “arrange” también puede ser enriquecida

Podemos definir asserts más expresivos para comprobar si los programas analizados funcionan correctamente

```
[PexMethod]
public void swapTest(IntPair target)
{
    int oldx = target.GetX();
    int oldy = target.GetY();
    target.swap();
    PexAssert.IsTrue(oldx == target.GetY() && oldy == target.GetX());
}
```

Tests de Unidad Parametrizados

Combinando asserts y assumes

Podemos combinar aserciones y suposiciones para enriquecer un mismo test de unidad paramétrico

```
public static IComparable maxComp(IComparable x, IComparable y)
{

    if (x.CompareTo(y) >= 0) return x;
    else return y;

}
```

Esto corresponde a un uso de contratos, para equipar el mecanismo de generación de tests

```
[PexMethod]
public IComparable maxComp(IComparable x, IComparable y)
{
    PexAssume.IsNotNull(x);
    PexAssume.IsNotNull(y);
    PexAssume.IsTrue(x.GetType()==y.GetType());

    IComparable result = BasicExample.maxComp(x, y);

    PexAssert.IsTrue(result.CompareTo(x)>=0);
    PexAssert.IsTrue(result.CompareTo(y)>=0);

    return result;
}
```

Sobre el Uso de PexAssert y PexAssume

PexAssert y PexAssume pueden usarse para anotar el código a analizar, pero

metodológicamente, conviene limitar su uso a tests de unidad parametrizados

"ensucian" el código a analizar con elementos sólo relevantes para el análisis

```
public static IComparable maxComp(IComparable x, IComparable y)
{
    PexAssume.IsNotNull(x);
    PexAssume.IsNotNull(y);
    PexAssume.IsTrue(x.GetType()==y.GetType());
    if (x.CompareTo(y) >= 0) return x;
    else return y;
}
```

Pex y Construcción de Objetos

Veamos una clase un poco más compleja que los ejemplos anteriores:

```
public class ArrayBasedList
{

    public static int maxSize = 10;
    private Object[] items;
    private int last = -1;

    public ArrayBasedList()
    {
        items = new Object[maxSize];
        last = -1;
    }

    public void add(Object item, int position)
    {
        ...
    }
```

Qué pasa si queremos generar tests para add?

Pex y Construcción de Objetos

Para permitir a Pex crear objetos y usarlos para testear métodos no estáticos, disponemos de las siguientes opciones:

- hacer públicos los atributos de la clase (metodológicamente incorrecto!)
- proveer constructores que alteren arbitrariamente los atributos de la clase, y hacerlos sólo disponibles en modo debugging
- crear “factory methods”, que permitan crear objetos usando constructores y otros métodos de la clase (e.g., insertando elementos, ...)

Aserciones de Corrección mediante Pex

Ya hemos visto anotaciones de programas a través de PexAssume y PexAssert. También vimos que su uso debería limitarse a la construcción de casos de tests.

Los conceptos de suposición y aserción en Pex se corresponden con los conceptos de precondición y postcondición, respectivamente, como contratos.

```
[PexMethod]
public IComparable maxComp(IComparable x, IComparable y)
{
    PexAssume.IsNotNull(x);
    PexAssume.IsNotNull(y);
    PexAssume.IsTrue(x.GetType()==y.GetType());

    IComparable result = BasicExample.maxComp(x, y);

    PexAssert.IsTrue(result.CompareTo(x)>=0);
    PexAssert.IsTrue(result.CompareTo(y)>=0);

    return result;
}
```

Contratos en el Código mediante Code Contracts

El valor de las aserciones de corrección de programas no se limita a la generación de casos de tests. Microsoft provee una librería para equipar programas con sus contratos respectivos, denominada Code Contracts.

Code Contracts permite:

- describir contratos de clases en el código de las mismas, como su especificación

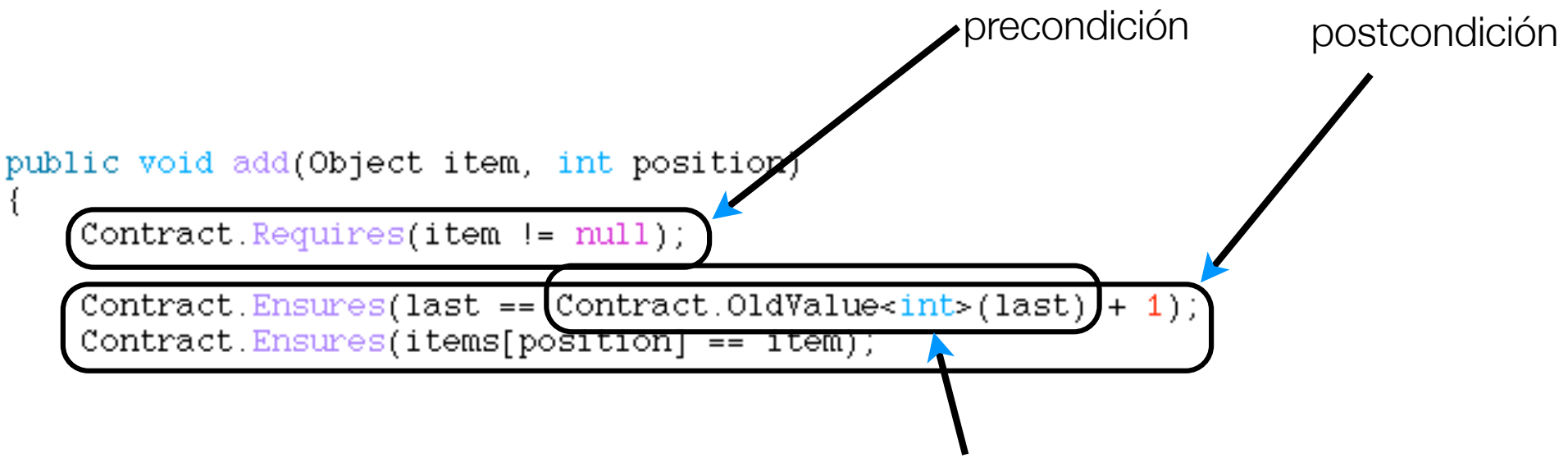
- interactuar con Pex para la generación de casos de tests

- comprobar dinámica y estáticamente la preservación de los contratos de programas

Además, ofrece un mayor poder expresivo que las aserciones provistas por Pex.

Precondiciones y Postcondiciones en Code Contracts

Veamos a través de un ejemplo la especificación de contratos mediante CodeContracts:



mayor poder expresivo que Pex: permite hacer referencia valores previos a la ejecución de la rutina, usar cuantificación, ...

Invariantes de Clase: Un Ejemplo

En Code Contracts, los invariantes de clase se definen mediante un método particular, que debe etiquetarse como invariante.

Combina aserciones (Contract.Invariant) con código

El invariante es verdadero en un estado si todas las cláusulas Invariant se satisfacen

```
[ContractInvariantMethod]
private void RepOK()
{
    Contract.Invariant(last >= -1 && last < items.Length);
    bool noNull = true;
    int i = 0;
    while (i <= last)
    {
        if (items[i] == null) noNull = false;
        i++;
    }
    Contract.Invariant(noNull);
}
```

Pex y Contratos

Pex aprovecha los contratos especificados con Code Contracts en dos sentidos:

- las precondiciones se usan para distinguir inputs válidos de inválidos

- Las postcondiciones y los invariantes de clase se usan como especificación de los resultados esperados al ejecutar métodos

    - las violaciones a los mismos se interpretan como bugs!

Poder Expresivo de Code Contracts

Algunas de los elementos que brindan mayor poder expresivo a Code Contracts, en comparación con aserciones Pex, son los siguientes

- Posibilidad de hacer referencia a los valores previos a la ejecución de la rutina:

```
Contract.OldValue<tipo>(expresión)
```

- Cuantificación universal:

```
Contract.ForAll (int fromInclusive, int toExclusive, Predicate<int>
predicate);
```

- Cuantificación existencial:

```
Contract.Exists (int fromInclusive, int toExclusive,
Predicate<int> predicate);
```

Cuantificadores en Code Contracts

El significado de los cuantificadores en Code Contracts es el siguiente:

- Cuantificación universal:

```
Contract.Requires(Contract.ForAll(0, args.Length, i => args[i] != null));
```

- Cuantificación existencial:

```
Contract.Requires(Contract.Exist(0, args.Length, i => args[i] == null));
```

Bibliografia

IntelliTest video tutorial:
https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/Intellitest

IntelliTest documentación:
https://docs.microsoft.com/en-us/visualstudio/test/generate-unit-tests-for-your-code-with-intellitest?view=vs-2019

Pex-White Box Test Generation:
https://www.microsoft.com/en-us/research/wp-content/uploads/2008/04/fulltext.pdf

Code Contract:
https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts