Técnicas de testing combinatorio y de mutación
Segunda parte

Macario Polo Usaola
Grupo Alarcos
Departamento de Tecnologías y Sistemas de Información
Universidad de Castilla-La Mancha
Ciudad Real, España

Contenidos

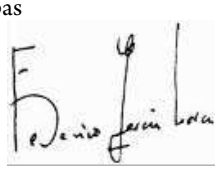
Mutación: conceptos importantes (I)

- Dado un programa o un sistema que vamos a probar, un *mutante* es una copia exacta de dicho programa o sistema, pero en el que introducimos un pequeño fallo
- El objetivo de la mutación consiste en escribir casos de prueba que encuentren esos fallos que se siembran, insertan o inyectan en los mutantes
- Si el fallo se encuentra, el mutante está *muerto*; en otro caso, el mutante está *vivo*

La casada infiel (fragmento)

Y que yo me la llevé al rio
creyendo que era mozuela,
pero tenia marido.

Fue la noche de Sanitago
y casi por conpromiso.
Se apagaron los faroles
y se encendieron los grillos.
En las ultimas esquinas
toqué sus pechos dormidos,
y se me avrieron de pronto
como ramos de jacintos.
El almidon de su enagua
me sonaba en el oido,
como una pieza de seda
rasgada por diez cuchiyos.
Sin luz de plata en sus copas
los árboles han cresido,
y un orizonte de perros
ladra muy lejos del rio.



Y que yo me la llevé al **río**
creyendo que era mozuela,
pero **tenía** marido.

Fue la noche de **Santiago**
y casi por **compromiso**.
Se apagaron los faroles
y se encendieron los grillos.
En las **últimas** esquinas
toqué sus pechos dormidos,
y se me **abrieron** de pronto
como ramos de jacintos.
El **almidón** de su enagua
me sonaba en el **oído**,
como una pieza de seda
rasgada por diez **cuchillos**.
Sin luz de plata en sus copas
los árboles han **crecido**,
y un **horizonte** de perros
ladra muy lejos del **río**.

Mutación: conceptos importantes (II)

- Aunque suene violento, el objetivo de las pruebas mediante mutación es *matar a tantos mutantes como sea posible*
- Tradicionalmente, la mutación se ha utilizado para evaluar la calidad de los test suites:
  - Efectivamente, si tenemos un conjunto de casos que encuentran *todos* los fallos inyectados, dicho conjunto es muy fiable
  - Desde luego, los fallos tienen que ser "buenos", no triviales
  - Además, si el test suite pasa por el SUT sin encontrar errores, tendremos una garantía muy alta de que éste es correcto/fiable/ …

Mutación: conceptos importantes (III)

- La calidad del test suite se mide mediante el *mutation score,* que viene dado por:

$$MS(P,T) = \frac{K}{(M - E)}$$

- … en donde P es el programa que se está probando; T es el test suite; K es el número de mutantes muertos; E es el número de mutantes *funcionalmente equivalentes*
- Un test que alcanza un *mutation score* de 1 se dice que es *mutation-adequate*

Mutación: conceptos importantes (IV)

- Un mutante funcionalmente o equivalente (o simplemente *equivalente*) es aquél que exhibirá en todo momento un comportamiento exactamente igual al del programa original.
- Es, por tanto, un mutante imposible de matar, y el *fallo* introducido no es tal, sino que representa una *optimización* o *desoptimización* del código

Mutación: conceptos importantes (V)

Version	Code

Original	int sum(int a, int b) {
	return a + b;
	}

Mutant 1	int sum(int a, int b) {
	return a - b;
	}

Mutant 2	int sum(int a, int b) {
	return a * b;
	}

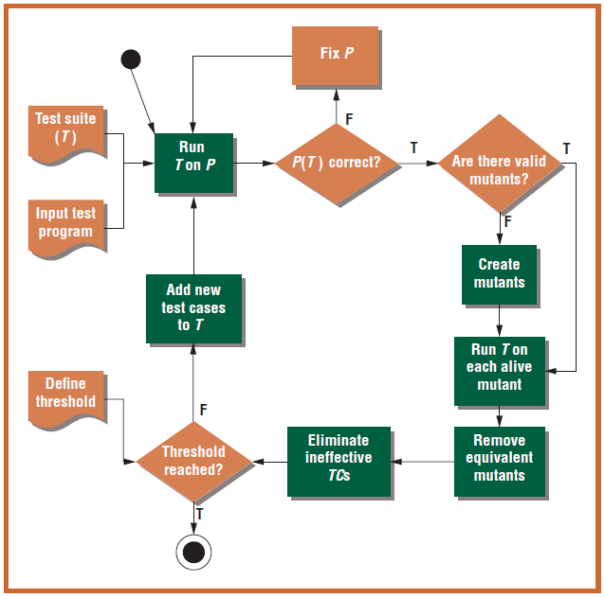
Mutant 3	int sum(int a, int b) {
	return a / b;
	}

Mutant 4	int sum(int a, int b) {
	return a + b++;
	}

Program versions	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
Original	2	0	-1	-2
Mutant 1	0	0	-1	0
Mutant 2	1	0	0	1
Mutant 3	1	Error	Error	1
Mutant 4	2	0	-1	-2

Mutación: conceptos importantes (y VI)

- Repaso:
  - Mutante
  - Mutante muerto
  - Mutante vivo
  - *Mutation score*
  - Mutante equivalente



Operadores de mutación

- Se utilizan *operadores de mutación* para insertar, introducir, inyectar o sembrar fallos artificiales en el código
- Cada copia (mutante) puede tener 1, 2 o *n* fallos insertados: se habla respectivamente de mutantes de orden 1, orden 2, orden *n*, o 1-*mutant*, 2-*mutant*, *n*-*mutant*
- Por lo general, los fallos introducidos son simples, y tratan de imitar los errores que cualquier programador podría cometer

10

Operadores de mutación

- Dos principios importantes:
  - Hipótesis del *programador competente:* los programadores escriben programas correctos o casi correctos.
  - Efecto acoplamiento (*coupling effect*): un *test suite* que detecta todos los fallos simples es suficientemente sensible para detectar errores más complejos. Comprobado experimentalmente.

Operadores de mutación *clásicos*

Operador	Descripción
ABS	Sustituir una variable por su valor absoluto
ACR	Sustituir una referencia variable a un array por una constante
AOR	Sustitución de un operador aritmético
CRP	Sustitución del valor de una constante
ROR	Sustitución de un operador relacional
RSR	Sustitución de la instrucción Return
SDL	Eliminación de una sentencia
UOI	Inserción de operador unario (p.ej.: en lugar de x, poner $-x$)

Operadores de mutación para objetos

Operador	Descripción
AMC (Access Modifier Change)	Reemplazo del modificador de acceso (por ejemplo: ponemos *private* en lugar de *public*)
AOC (Argument Order Change)	Cambio del orden de los argumentos pasados en la llamada a un método (p.ej.: en lugar de *Persona p=new Persona("Paco", "Pil")* poner *Persona p=new Persona("Pil", "Paco")*
CRT (Compatible Reference Type Replacement)	Sustituir una referencia a una instancia de una clase por una referencia a una instancia de una clase compatible (p.ej.: en vez de poner *Persona p=new Empleado()*, poner *Persona p=new Estudiante()*).
EHR (Exception Handgling Removal)	Eliminación de una instrucción de manejo de excepciones
HFA (Hiding Field variable Addition)	Añadir en la subclase una variable con el mismo nombre que una variable de su superclase
MIR (Method Invocation Replacement)	Reemplazar una llamada a un método por una llamada a otra versión del mismo método
OMR (Overriding Method Removal)	Eliminar en la subclase la redefinición de un método definido en una superclase
POC (Parameter Order Change)	Cambiar el orden de los parámetros en la declaración de un método (p.ej.: poner *Persona(String apellidos, String nombre)* en vez de *Persona(String nombre, String apellidos)*
SMC (Static Modifier Change)	Añadir o eliminar el modificador *static*

13

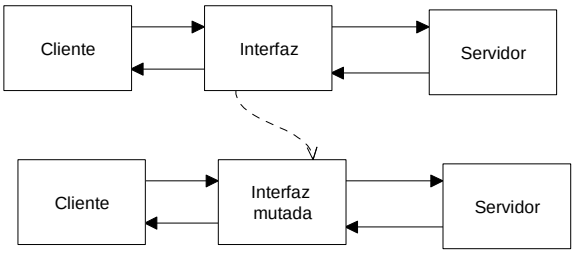
Operadores de mutación para componentes

- Al no disponer del código, tratamos de introducir otro tipo de errores para verificar nuestro *test suite* y el comportamiento del componente (quien dice componente, dice subsistema, librería, servicio web, sistema externo, etc.)
- Algunos operadores:
  - Reemplazar *inout* por *out*, o *out* por *inout*
  - Intercambiar parámetros de tipos compatibles
  - "Jugar" con un parámetro (operador *twiddle*)
  - Pasar cero como parámetro en los valores numéricos
  - Pasar *null* como parámetro en valores complejos

14

Operadores de mutación para componentes

- Criterios de cobertura en este caso:
  - Llamadas a métodos o servicios
  - Excepciones lanzadas
  - Combinación de ambos (llamar a cada método, de manera que cada método lance todas sus posibles excepciones)



15

Reducción de costes

- El número de mutantes que potencialmente puede generarse es elevadísimo:
  - En 10 programas escritos en Fortran-77, con entre 10 y 48 sentencias ejecutables, se generaron entre 183 y 3010 mutantes
  - En 11 programas con una media de 43,7 LDC, se generaron 3211 mutantes
- El proceso de testing con mutación tiene 3 pasos importantes:
  - Generación de mutantes
  - Ejecución del *test suite* contra los mutantes
  - Análisis de resultados

16

Reducción de costes en la generación: mutación selectiva (I)

- El objetivo es reducir el número de mutantes sin perder eficiencia
- Con un número muy grande de mutantes:
  - Se generarán muchos, con lo que este paso puede ser lento
  - Habrá muchas versiones del programa para ejecutar el *test suite,* con lo que este paso puede ser muy leeennnnto
  - Habrá que dedicar mucho tiempo a buscar los mutantes equivalentes (en torno a un 20% de los generados), o bien "pasar" de ellos y calcular el *mutation score* sin considerarlos

17

Mutación selectiva (II)

- Hay dos estrategias de mutación selectiva:
  - Selección de un conjunto aleatorio de mutantes (20-25%). Muy eficiente: un *test suite* que mata todos matará con probabilidad casi todos
  - Selección de los mejores operadores de mutación, que son los siguientes:
    - AOR (reemplazo de operador aritmético)
    - SDL (*statement deletion,* eliminación de sentencias)
    - ROR (reemplazo de operador relacional)
    - UOI (*unary operator insertion,* inserción de operador unario)
    - LCR (*logical connector repacement*, reemplazo de conector lógico)

18

Mutación selectiva (y III)

- Distribución de mutantes equivalentes por operador, en la herramienta MuJava

Program	# of 1-order mutants	Total equivalent mutants	Equivalent mutants per mutation operator				
			AOIS	AOIU	ROR	LOI	COR
Bisect	63	19	14	3	2	0	0
Bub	82	12	9	0	3	0	0
Find	179	0	0	0	0	0	0
Fourballs	212	44	42	1	0	1	0
Mid	181	43	38	2	1	2	0
TriTyp	309	70	54	8	1	4	3
			83,5%	7,4%	3,7%	3,7%	1,6%

19

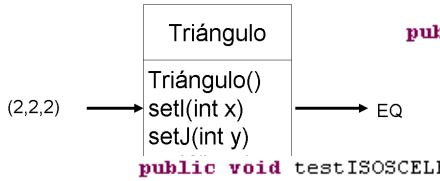
Reducción de costes en la ejecución (I)

- Matriz de muertos:

Reducción de costes en la ejecución (II)

- En general se utiliza *mutación fuerte* (*strong mutation*).
  - Requiere las condiciones *RIP* (***R**equiem **I**n **P**acem*) para matar al mutante:
    - ***R**eachability* ("alcanzabilidad" o *posibilidad de alcance*): la sentencia mutada debe ser alcanzable.
    - ***I**nfection* (infección): la sentencia mutada debe producir un estado erróneo (distinto del estado en esa ubicación en el programa original).
    - ***P**ropagation* (propagación): el estado erróneo debe propagarse hasta el final de la ejecución del caso de prueba, de manera que sea observable y comparable con el estado del programa original. Se requiere la escritura de un oráculo suficientemente completo.

Reducción de costes en la ejecución (III)




```
public void testEQ() {
    Triangulo t=new Triangulo();
    t.setI(2);
    t.setJ(2);
    t.setK(2);
```

```
public void testISOSCELES() {
    Triangulo t=new Triangulo();
    t.setI(2);
    t.setJ(2);
```

```
public void testNT() {
    Triangulo t=new Triangulo();
    t.setI(2);
    t.setJ(2);
    t.setK(4);
    assertTrue(t.getTipo()==
        Triangulo.NO_TRIANGULO);
}
```

```
public void testESC() {
    Triangulo t=new Triangulo();
    t.setI(2);
    t.setJ(3);
    t.setK(4);
    assertTrue(t.getTipo()==
        Triangulo.ESCALENO);
}
```

Reducción de costes en la ejecución (IV)

- Una estrategia diferente es la mutación débil *(weak mutation),* que requiere solamente las condiciones de *Reachability* e *Infection*
- El estado del programa original y el del mutante se comparan en un punto determinado después de la instrucción mutada:
  - Si el estado distinto, el mutante se considera muerto
  - Si los estados son iguales, se da por vivo y se detiene la ejecución del caso de prueba

Reducción de costes en la ejecución (V)

- Otra estrategia es *Functional Qualification*, en la que se reutilizan casos de prueba de caja negra (tipo JUnit)
  - El oráculo (instrucciones *assert)* es lo que se utiliza para determinar si el mutante está vivo o muerto
- Realmente es un tipo de mutación fuerte

Reducción de costes en la ejecución (y VI)

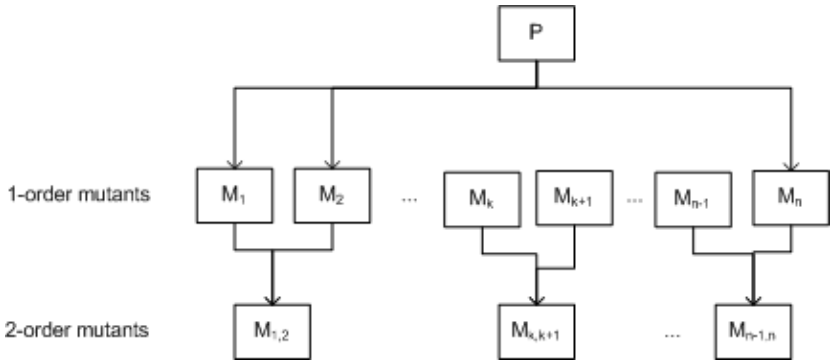
- Otra estrategia diferente es la mutación débil flexible *(Flexible Weak Mutation)*
- En FWM, el estado de los dos programas se compara constantemente
  - En cuanto se detecta un cambio, el mutante se da por muerto y se detiene la ejecución
- Es la técnica que implementa Bacterio, que veremos después, y es adecuada para sistemas multiclases

Reducción de costes en el análisis de resultados (I)

- El problema principal reside en los mutantes equivalentes, que constituyen "ruido". Formalmente, el problema de la detección de mutantes equivalentes es indecidible.
- Algunos autores hablan de un tiempo medio de 15 minutos para detectarlos
- Entonces, lo que se hace normalmente es:
  - O bien utilizar mutación selectiva
  - O bien considerar de antemano que en torno a un 20% de mutantes serán equivalentes
  - O bien una combinación de los dos puntos anteriores

Reducción de costes en el análisis de resultados (II)

- Otra posibilidad consiste en utilizar mutación de orden *n* (es decir, introducir más de un fallo en cada mutante)



27

Reducción de costes en el análisis de resultados (III)

- Combinación de dos mutantes en un mutante de orden 2

M_i	M_j	$M_{i,j}$
Equivalent	Equivalent	Equivalent
Equivalent	Non-equivalent	Non-equivalent
Non-equivalent	Equivalent	Non-equivalent
Non-equivalent	Non-equivalent	Non-equivalent (very probably)

- Obviamente:

$$T \text{ is mutation - adequate for } M \Rightarrow T \text{ is mutation - adequate for } M'$$

- ¿Se cumple lo recíproco?

28

Reducción de costes en el análisis de resultados (IV)

- Algoritmos para combinación de mutantes
  - *LastToFirst*
  - *DifferentOperators*
  - *RandomMix*

AOIS_1	AOIS_35	AOIS_56	AOIU_11	AORB_1	ROR_1
AOIS_10	AOIS_37	AOIS_58	AOIU_12	AORB_10	ROR_10
AOIS_12	AOIS_38	AOIS_59	AOIU_13	AORB_3	ROR_4
AOIS_13	AOIS_39	AOIS_60	AOIU_2	AORB_5	ROR_6
AOIS_14	AOIS_4	AOIS_71	AOIU_3	AORB_7	
AOIS_15	AOIS_40	AOIS_72	AOIU_4		
AOIS_16	AOIS_41	AOIS_73	AOIU_6		
AOIS_17	AOIS_43	AOIS_74	AOIU_7		
AOIS_18	AOIS_44	AOIS_75	AOIU_8		
AOIS_19	AOIS_45	AOIS_76	AOIU_9		
AOIS_20	AOIS_47	AOIS_77			
AOIS_25	AOIS_48	AOIS_78			
AOIS_27	AOIS_5	AOIS_79			
AOIS_3	AOIS_52	AOIS_80			
AOIS_31	AOIS_54				

29

Reducción de costes en el análisis de resultados (V)

Program	LOC	# of 1-order mutants	Equivalent 1-order mutants		# of test cases
			Number	%	
Bisect	31	63	19	30.15%	25
Bub	54	82	12	14,63%	256
Find	79	179	0	0.00%	135
Fourballs	47	212	44	20.75%	96
Mid	59	181	43	23.75%	125
TriTyp	61	309	70	22.65%	216
			Mean:	18,66%	

Program	LastToFirst			DifferentOperators			RandomMix		
	# mutants (*)	Equivalent		# mutants (*)	Equivalent		# mutants (*)	Equivalent	
		#	%		#	%		#	%
Bisect	32 (50,8%)	5	15,63	44 (69,8%)	5	11,36	32 (50,8%)	2	6,25
Bub	41 (50%)	0	0	44 (53,7%)	0	0	40 (48,8%)	1	2,5
Find	90 (50,3%)	0	0	97 (54,2%)	0	0	89 (49,7%)	0	0
Fourballs	107 (50,4%)	5	4,67	128 (60,4%)	6	4,68	106 (50%)	7	6,60
Mid	91 (50,3%)	8	8,79	110 (60,8%)	4	3,63	91 (50,3%)	7	7,69
TriTyp	155 (50,2%)	7	4,51	168 (54,4%)	11	6,54	155 (50,2%)	9	5,80
	Means:		5,6%			4,4%			4,8%

30

Reducción de costes en el análisis de resultados (VI)

Mutants	Test cases 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
AOIS_1	1	1		1	1		1				1	1		1	1					1					
AOIS_10	1		1		1	1		1	1	1	1		1		1	1		1	1	1	1		1	1	1
AOIS_12	1		1		1	1		1	1	1	1		1		1	1		1	1	1	1		1	1	1
AOIS_13			1		1	1	1	1	1	1	1	1	1		1	1	1	1		1	1	1	1	1	1
AOIS_14	1		1	1	1	1		1	1	1	1		1	1	1	1	1	1	1	1	1		1	1	1
AOIS_15			1		1	1	1	1																	
AOIS_16	1		1	1	1	1		1																	
AOIS_17			1		1	1	1	1																	
AOIS_18	1		1	1	1	1	1	1																	
AOIS_19	1	1		1	1																				
AOIS_20	1	1		1	1																				
AOIS_25	1	1		1	1		1																		
AOIS_27	1	1		1	1																				
AOIS_35		1		1	1																				
AOIS_37		1		1	1																				
AOIS_38	1	1		1	1																				

Bisect	
KM	TC
17	3
18	5
20	1
22	1
23	1
24	3
25	1
26	1
28	1
30	1
31	1
35	1
37	2
40	2
42	1

Bub	
KM	TC
1	174
2	1
25	1
26	1
29	9
30	3
59	5
61	4
62	3
63	5
64	5
65	8
66	11
67	19
68	6

Find	
KM	TC
2	1
3	75
4	20
178	81

Fourballs	
KM	TC
43	12
46	6
49	14
50	6
52	12
53	8
54	8
55	10
56	2
59	10
60	4
67	4

Mid			
KM	TC	KM	TC
11	1	40	3
14	4	41	4
17	2	42	1
19	3	43	5
20	1	45	2
21	1	46	1
22	1	47	3
23	1	48	1
25	2	49	4
26	3	50	1
28	4	51	3
29	3	52	3
30	3	53	2
31	2	54	2
32	6	55	7
33	1	56	3
34	3	57	5
35	1	59	2
36	4	60	5
37	6	62	2
38	9	64	1
39	3	67	1

TriTyp			
KM	TC	KM	TC
0	56	86	2
3	48	88	1
4	48	89	1
22	2	91	2
23	4	94	3
39	2	96	1
44	2	97	1
48	2	100	1
51	4	101	1
57	4	103	1
62	4		
66	1		
68	1		
69	4		
70	4		
71	1		
73	1		
75	1		
77	6		
78	2		
84	3		
85	2		

31

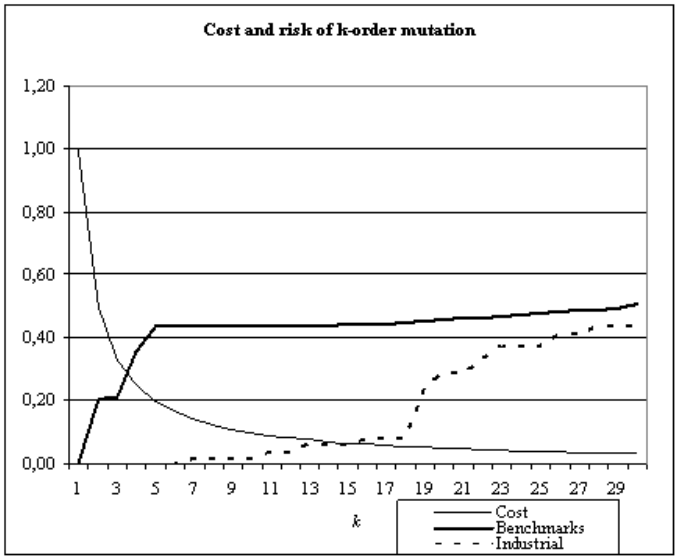
Reducción de costes en el análisis de resultados (VII)

Program	# of sentences	WMC	# of methods	# of 1-order mutants	# of available test cases	1-order mutans killed
Ciudad	177	65	23	203	37	92%
IgnoreList	26	8	3	27	6	85%
PluginTokenizer	157	27	16	94	14	31%

Program	1-order mutants		2-order mutants		
	#	% killed	#	% reduction	% killed
Ciudad	203	92%	110	54,18%	94%
IgnoreList	27	85%	17	62.96%	88%
PluginTokenizer	94	31%	48	51,06%	50%

Reducción de costes en el análisis de resultados (y VIII)

$$risk(k) = 1 - \frac{\text{\# of test cases killing } k \text{ mutants or more}}{\text{total \# of test cases}}$$

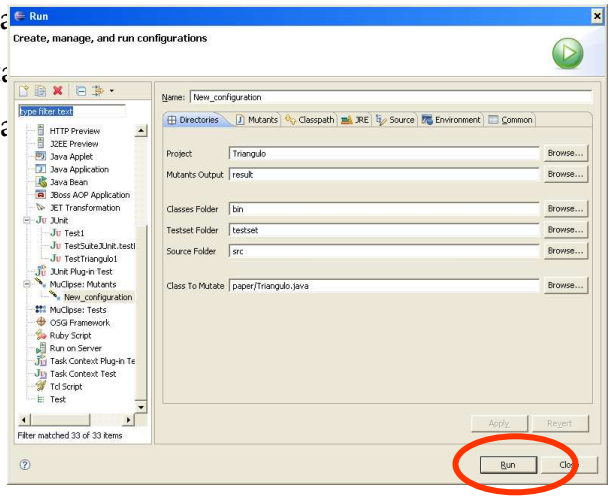


Herramientas

- Hay muchas, sobre todo para Java (*Jester, JavaMut, muJava, MuClipse, Jumble, Javalanche, ExMan, Mugamma, AjMutator, Judy*); algunas para C y C++ (*PleTest C++, Certitude*)
- Veamos cómo funciona MuClipse

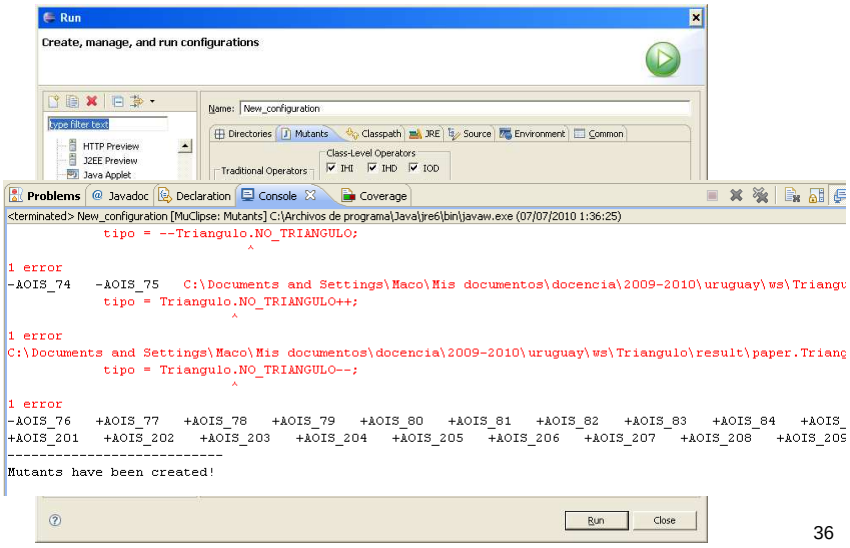
MuClipse (I)

- Los pasos eran:
  - Genera
  - Ejecut
  - Analiz



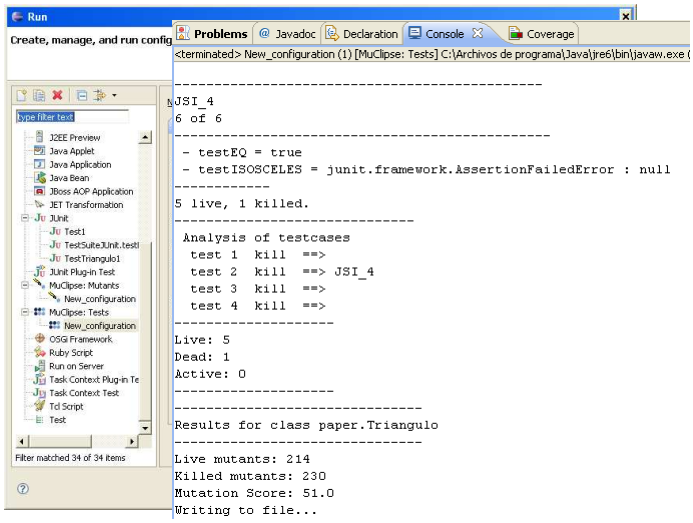
MuClipse (II)

- Generación de mutantes



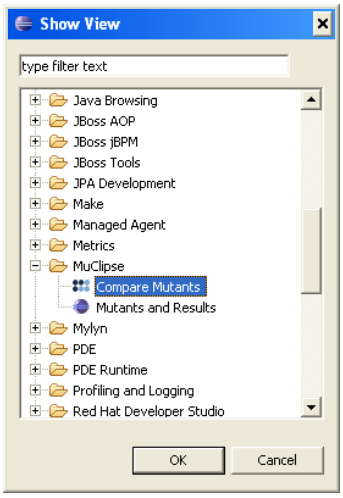
MuClipse (III)

- Ejecución de casos



MuClipse (y IV)

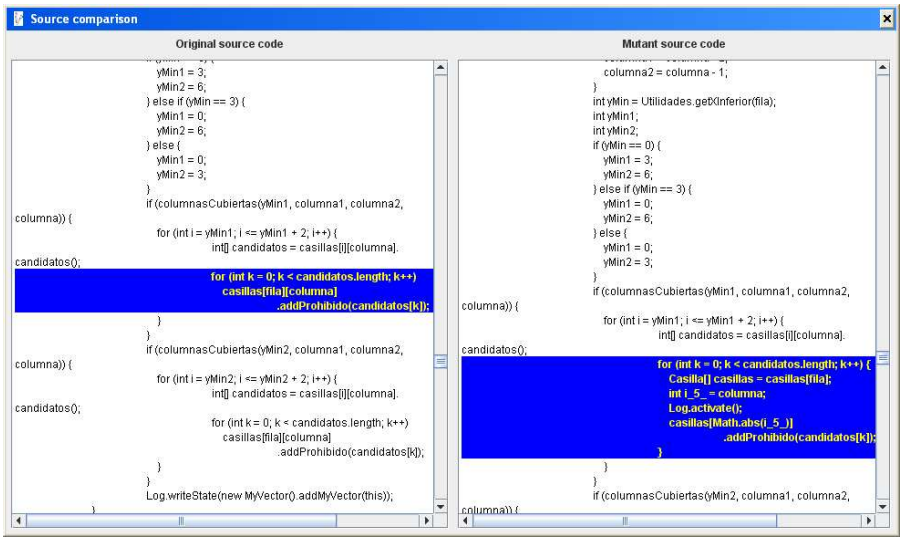
- Análisis de resultados



Bacterio

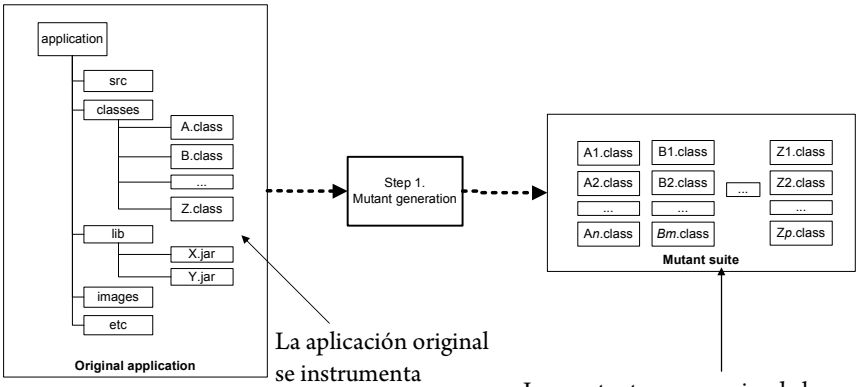
- Es una herramienta que hemos desarrollado en el Grupo Alarcos de la UCLM
- Permite dar soporte a las pruebas de aplicaciones Java complejas usando mutación
- Incorpora:
  - Mutación a nivel de *bytecode* (no usa código fuente)
  - Mutación selectiva (de operadores y de selección aleatoria de mutantes) y de orden *n*
  - *Flexible Weak Mutation*
  - Captura y reejecución, testing exploratorio
  - Descompilador para detección de mutantes equivalentes

Bacterio: detección de mutantes equivalentes



Bacterio

- Generación de mutantes (I)



La aplicación original se instrumenta

Los mutantes son copias de la versión original instrumentada (es decir, tb. están instrumentados)

41

Bacterio

```
protected void putNumbers(String s){

  i=0;
  …

  return;
}
```

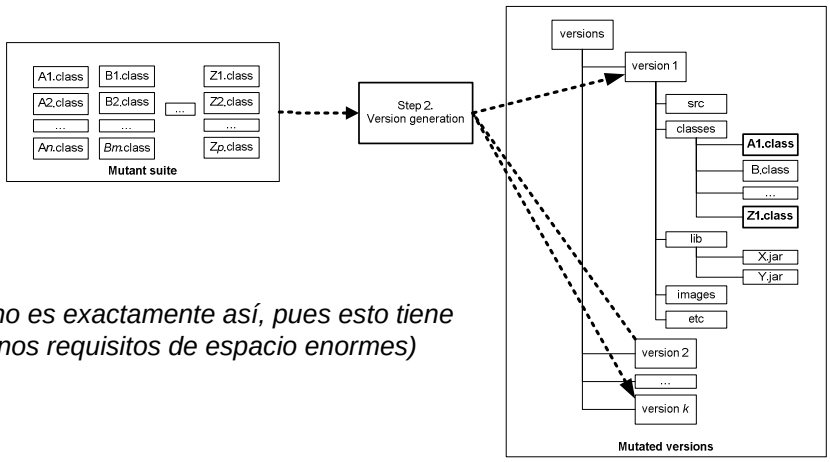
```
protected void putNumbers(String s){
  try{
    Log.writePreState(this);
    i=0;
   …
    Log.writePostState(new
     Vector().add(s).add(this));
    Return;
  }
  catch(Thowable t){

    Log.writePostState(this);
    throw t;
  }
}
```

42

Bacterio

- Generación de mutantes (II)



(no es exactamente así, pues esto tiene unos requisitos de espacio enormes)

43

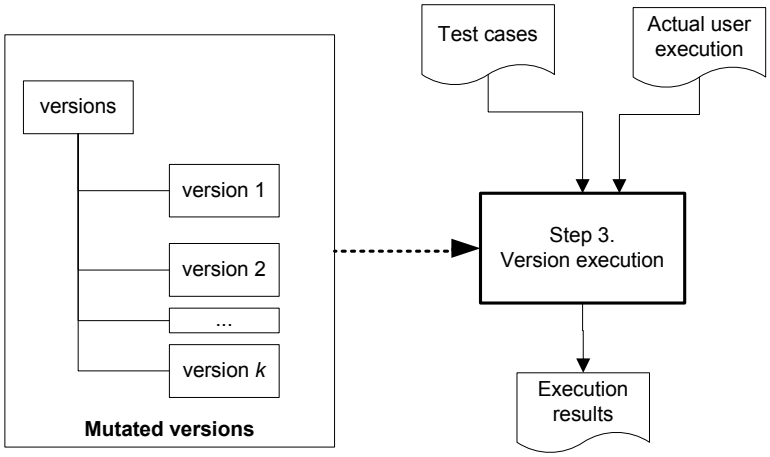
Bacterio

- Generación de mutantes (y III)
- Operadores:
  - Swap parameters
  - Parameter increment/decrement
  - Nullify
  - Throw exceptions
  - AOR
  - ABS
  - ROR
  - UOI

44

Bacterio

- Ejecución de casos



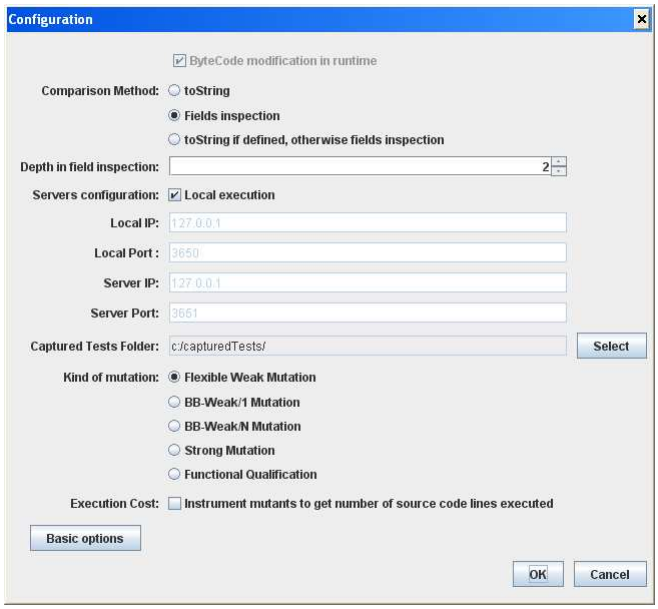
45

Bacterio

- La FWM se consigue gracias a la instrumentación
- Cada vez que se llama a *Log.writeX(…)*, se compara la traza del original con la del mutante
- Hay tres formas de registrar la traza:
  - *Field inspection* (mediante reflexión)
  - *toString()*
  - *toString()* si existe, *field inspection* en otro caso

46

Bacterio: opciones de configuración



Bacterio: generación de mutantes

Humm… mejor ver una demo

Bacterio VS Muclipse

			Mutation score (%)		
				Bacterio	
Subject	Tool used	# of tests	Muclipse	Functional qualification	Flexible weak mutation
Andrea	Bacterio	17	2.1	2.43	34.70
César	Bacterio	11	7.53	8.79	28.37
Javi	Bacterio	8	3.31	4.33	32.00
Laura	Bacterio	17	8.43	10.55	55.72
Moisés	Bacterio	6	10.54	11.9	64.44
Bea	Muclipse	12	5.01	6.22	67.86
Carlos	Muclipse	15	6.92	8.52	69.57
Luis	Muclipse	7	12.04	14.61	57.94
Nacho	Muclipse	11	6.77	7.84	31.28
Ricardo	Muclipse	37	8.43	10.28	63.93

49

Análisis de *usabilidad*

	S1	S2	S3	S4	S5	Mean
Communication						
Tasks are understandable	4	5	4	4	2	3.8
Tasks are easy to remember/execute	5	5	4	5	3	4.4
The language used is adequate	5	5	4	4	3	4.2
The users have the control of the application	2	5	4	4	2	3.4
In the context of testing and mutation, the application is intuitive	3	4	4	3	3	3.4
Consistence						
Presentation is consistent	4	5	4	5	3	4.2
Feedback						
Information about the execution of tasks is available	3	5	5	5	2	4
Users know the task under execution and the required information to execute it	4	3	4	4	2	3.4
Error control						
There are adequate error messages	1	1	-	4	1	1.4
Attractiveness						
Information is well presented on the screen	3	5	4	4	2	3.6
Information is clear and legible	4	5	4	4	2	3.8
The tool is easy to use in daily tasks	5	5	4	5	3	4.4

50

Casos de prueba redundantes

- Un caso de prueba *a* es redundante con respecto a otro caso *b* cuando, al ejecutarlo:
  - O bien *a* recorre las mismas *porciones de código* que *b*,
  - o bien las *porciones de código* que recorre *a* son un subconjunto de las *porciones de código* que recorre *b*
- Al habla de "porciones de código" estamos referencia a la cobertura alcanzada
- Por ejemplo, si *a* mata los mismos mutantes que *b*, o los que mata son un subconjunto de los que mata *b*, entonces *a* es redundante respecto a *b*

Reducción del conjunto de casos

- La determinación del *test suite* de cardinal mínimo es un problema NP-completo (no resoluble en tiempo polinomial)
- Por ello, se utilizan algoritmos voraces:
  - Añadimos al test suite el caso que más mutantes mata
  - Eliminamos los mutantes muertos
  - Añadimos al test suite el siguiente caso que mate más
  - Eliminamos los muertos
  - Continuamos así hasta que no queden casos que maten más mutantes

Reducción del conjunto de casos

Mutant	Each *X* represents that the tc_i test case has killed the m_j mutant					
	*tc*1	*tc*2	*tc*3	*tc*4	*tc*5	*tc*6
*m*1	X	X				
*m*2	X	X			X	
*m*3		X				X
*m*4			X			X
*m*5			X			X
*m*6			X			X
*m*7						X

- En Bacterio, ejecutaríamos con la opción *Full tests*

Ejemplo

Total killed mutants

Total versions	Total Mutants	abs	uoi	ror	aor	inc	nuf	dec	swa
308/511	308/510	0/95	83/95	141/223	56/64	11/11	2/6	9/10	6/6

Percentage

Total versions	Total Mutants	abs	uoi	ror	aor	inc	nuf	dec	swa
60.27%	60.39%	0.0%	87.36%	63.22%	87.5%	100.0%	33.33%	90.0%	100.0%

Killing matrix

Version	tests.Test1-testGoForward	tests.Test1-testGoBack
/version495	X	X
/version496	X	X
/version497	X	X
/version498	X	X
/version499	X	X
/version5		X
/version50		
/version500	X	X
/version501	X	X
/version502		
/version503		

Recapitulación

- Criterios de cobertura.
  - Es una condición de parada del proceso de prueba. Por ello, se establece el umbral deseado *antes* de probar.
  - Sirven principalmente para tres cosas:
    - Determinar las porciones del sistema que no se han probado
    - Escribir nuevos casos de prueba para recorrer esas áreas inexploradas
    - Conocer cuantitativamente el valor de la cobertura que, indirectamente, puede ser un valor muy adecuado para determinar la calidad o la fiabilidad del sistema
- Para código, tablas de decisión, máquinas de estado que incluyan guardas, etc., es muy útil el criterio *MC/DC*

55

Recapitulación

- Valores de prueba: los que el tester considera "interesantes". Proceden de técnicas como clases de equivalencia, valores límite, conjetura de errores, experiencia, entrevistas, etc.
- Debemos ser capaces de conocer cuantitativamente el grado en que usamos los valores de prueba. Para ello, tenemos criterios de cobertura de valores:
  - *Each choice*
  - *Pairwise*
  - *N-wise*

56

Recapitulación

- Dado un criterio de cobertura para valores, disponemos de diferentes estrategias de combinación para alcanzarlos (algoritmos)
  - *Each use*
  - *AETG*
  - *All combinations*
  - *Antirandom*
  - *Comb*
  - …

Recapitulación

- Conceptos importantes en mutación:
  - Mutante
  - Mutante muerto
  - Mutante vivo
  - *Mutation score*
  - Mutante equivalente
  - Mutante de orden *n*
  - Tipos de ejecución de casos contra mutantes (*strong mutation, weak mutation, flexible weak mutation, functional qualification*)
  - Matriz de muertos

Recapitulación

- Proceso de la mutación:
  - Generación de mutantes
  - Ejecución de casos
  - Análisis de resultados
- Técnicas de reducción de costes:
  - Mutación de orden *n*
  - Mutación selectiva
  - Selección aleatoria de mutantes

Recapitulación

- Utilicemos conjuntamente los valores interesantes y las estrategias de combinación para obtener buenos casos de prueba (buenos *test suites*)
- Ejecutémoslos con una herramienta de mutación
- Si no matamos suficientes mutantes, enriquecemos el *test suite* con más casos que los vayan matando
- Herramientas:
  - MuClipse
  - Bacterio

Técnicas de testing combinatorio y de mutación

Macario Polo Usaola
Grupo Alarcos
Departamento de Tecnologías y Sistemas de Información
Universidad de Castilla-La Mancha
Ciudad Real, España