From Constraint-based Refactoring

Friedrich Steimann
Fernuniversität in Hagen
October 27, 2010

to Constraint-based Mutant Generation

Friedrich Steimann
Fernuniversität in Hagen
October 27, 2010

Constraint-based refactoring

Refactoring

Refactoring is the **behaviour-preserving** change of (a program's) design.

Move Class (to another package)

```
package a;

class A {
   B b;
}

class B {
}
```

```
package b;
```

access violation

Move Class (to another package)

```
package a;
class A {
  public void m(Object o)
    { … }
  void m(String s) { … }
}
class B {
  {
    new A().m("a");
  }
}
```

```
package b;
```

change of static binding

Move Class (to another package)

```
package a;
public class A {
  public void m() {…}
  public void n()
    { m(); }
}
class B extends A {
  void m() {…}
}
new B().n();
```

```
package b;
class B extends A {
  public void m() {…}
}
class C …
  void …
}
new C().n();
```

change of …

subtyping error

Constraint-based refactoring

Basic program elements

A program *P* consists of
- a set *D* of **declared entities** and
- a set *R* of **references** to elements of *D*.

Every $d \in D$ has
- a **declared accessibility**, $d.\alpha$
- and a **location**, $d.\lambda$, in *P*.

Every $r \in R$ has
- a location, $r.\lambda$, in *P*.

Constraint variables

- $d.\alpha$, the declared accessibility of a declared entity
  - $d.\alpha \in A = \{absent, private, package, protected, public\}$
    $absent < private < package < protected < public$
- $d.\lambda$, the location of a declared entity
  - $d.\lambda \in L$ (≅ the bodies of the types of *P*)
- $r.\lambda$, the location of a reference
  - $r.\lambda \in L$
- $\alpha: L \times L \rightarrow A$, a function computing required accessibility

Constraint variables example

```
class A {
  double d;
}

class B {
  A a;
  { a.d = 1.0; }
}
```

- d α = *package*
- d λ = A
- A λ = B
- a α = *package*
- a.d λ = B

Constraint rules

- accessing:

$$\frac{binds(r, d)}{d.\alpha \geq \alpha(r.\lambda, d.\lambda)}$$

$$\frac{binds(\text{a.d}, \text{d})}{d.\alpha \geq \alpha(\text{a.d}.\lambda, \text{d}.\lambda)}$$

$d.\alpha \geq \alpha(\text{B}, \text{A}) = package$

```
class A {
  double d;
}

class B {
  A a;
  { a.d = 1.0; }
}
```

Constraint rules

```
class A {
  private double d;
  B b;
  { b.d = 1.0; }
}

class B extends A
  {}
```

- inheritance:

$$\frac{binds(r, d) \quad rtype(r, t)}{d.\alpha \geq \alpha(t, d.\lambda)}$$

$$\frac{binds(\text{b.d}, d) \quad rtype(\text{b.d}, \text{B})}{d.\alpha \geq \alpha(\text{B}, d.\lambda)}$$

$d.\alpha \geq \alpha(\text{B}, \text{A}) =$ *package*

not in C#!

Constraint rules

- subtyping:

$$\frac{\text{overrides}(d', d) \vee \text{hides}(d', d)}{d'.\alpha \geq d.\alpha}$$

- dynamic binding:

$$\frac{\text{overrides}(d', d)}{d.\alpha \geq \alpha(d'.\lambda, d.\lambda)}$$

Constraint rules

- arrays:
  $T[\,].\alpha = T.\alpha$
- multiple declarations such as `int i, j;`:
  $i.\alpha = j.\alpha$
- admissible access modifiers:
  $d.\alpha \in A(d, d.\lambda)$
- main methods, imported elements, several top-level types in a compilation unit, …

Basic idea

- Generate accessibility constraints from program as is.
  - The variable assignments derived from the program as is present a solution of the constraint set.
- Assign variables new values that reflect the intended refactoring.
- Check whether constraint system is still solved; if not **and user permits**, try to solve it.
- Write back changed values, or deny refactoring.

Example: Move Class

```
package a;

class A {
  B b;
}

class B {
}
```

```
package b;

public class B {
}
```

$B.\alpha \geq \alpha$(a.A, b) = *pabkage*

Example: Pull Up Method

```
package a;

public class A {
  public void m(Object o) {…}
  void m(String s) {…}   // struck through
}
```

```
package b;

class B extends A {
  void n() {
    m("abc");
  }
}
```

m(Object)$.\alpha \geq \alpha$(b.B, a.A) = *privatected*

m(String)$.\alpha < \alpha$(b.B, a.A) = *privatected*

What we have

- program transformations …
  - … resulting in a compiling program
  - … and **not altering** the program's behaviour

Steimann, Thies: *ECOOP* (2009)

Constraint-based mutant generation

Steimann, Thies: *ICSE* (2010)

Mutant generation is non-trivial

```
class A {
  private void
    m(String s) {…}
}
class B extends A {
  private void m(Object o) {…}
  void main() {
    new B().m("a");
  }
}
```

Only **0.03%** of the compiling **access modifier change** mutants actually change the program's behaviour! [Offutt'06]

Will it compile?

Will it likely change the program's behaviour? (be non-equivalent)

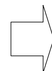
Will multiple mutants point to same missing test case? (redundant mutants)

What we need

- program transformations …
  - … resulting in a compiling program
  - … and **not altering** the program's behaviour
- program transformations …
  - … resulting in a compiling program
  - … and **altering** the program's behaviour

Constraint-based mutant generation

```
class A {
  private void m(String s) {…}
  void m(Object o){…}
}

class B {
  void main() {
    new A().m("abc");
  }
}
```



$A.\alpha \neq$ protected

$A.\alpha \geq \alpha(B.\lambda, A.\lambda)$

…

$A.m(String).\alpha < \alpha(B.\lambda, A.\lambda)$

Constraint-based mutant generation

```
class A {
  private void m(String s) {…}
  void m(Object o){…}
}

class B {
  void main() {
    new A().m("abc");
  }
}
```

$A.\alpha \neq$ protected

The Class A must not be declared with protected accessibility level.

Otherwise: Compiler error

Constraint-based mutant generation

```
class A {
  private void m(String s) {…}
  void m(Object o){…}
}

class B {
  void main() {
    new A().m("abc");
  }
}
```

A.α ≠ protected

A.α ≥ α(B.λ, A.λ)

…

A.m(String).α < α(B.λ, A.λ)

Constraint-based mutant generation

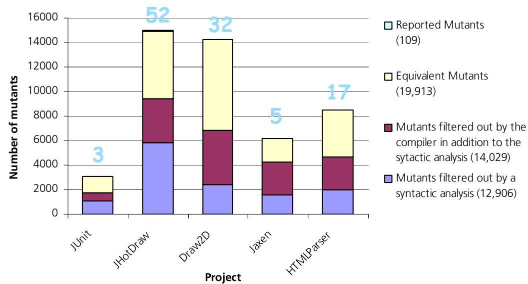
```
class A {
  private void m(String s) {…}
  void m(Object o){…}
}

class B {
  void main() {
    new A().m("abc");
  }
}
```

A.m(String) must remain inaccessible in B.

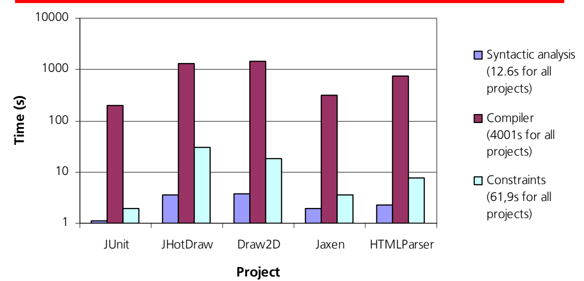
Otherwise: Change of binding

A.m(String).α ≮ ≥ α(B.λ, A.λ)

Evaluation: Mutant filtering



Evaluation: Runtime



Evaluation: Completeness

- complete **iff** the constraint rules model the language specification completely
- constraint rules tested by thousands of automatic refactoring applications on open source projects and automatic check for unchanged behaviour
  - with the help of the compiler
  - with the help of the provided JUnit-Tests

use tests to check
the completeness of an approach
that checks the completeness
of (other) tests

Outlook

Incorporating other constraint rules

Type Constraints [Tip et al. 2003]

```
class A {
  String m(A a) {…}
}

class B extends A {
  String m(A a) {…}
}           B

   … new B().m(new A())
```

A.m(A) overrides B.m(A) so the parameter types of both methods ~~must~~ be equal
must not

$$[Param(B.m,0)] \neq [Param(A.m,0)]$$

Enhancing Mutant Reports

```
public class A {
  private String m(String s) {
    return "a";
  }
}

public class B extends A {
  String m(Object o) {
    return "b";
  }
}

  …. new B().m("abc");
```

```
@Test
public void testM() {
  // TODO Auto-generated
  // test object
  B b = null;

  // TODO Auto-generated
  //test parameter
  String param1 = null;

  // TODO Auto-generated
  // test expectation
  String expected = null;

 Assert.assertEquals(expected,
  b.m(param1));
}
```


Thank you!
Questions?