Behavioral Program Queries using Logic Source Code Templates

Coen De Roover ▲ Johan Brichau ♦ Carlos Noguera ▼

▲ Programming Technology Lab - Vrije Universiteit Brussel - Belgium
♦ Département d'Ingénierie Informatique - Université catholique de Louvain - Belgium
▼ INRIA - Futurs, Jacquard Project - France
Program Queries

identify elements implementing

- high-level design
- refactoring preconditions
- run-time bugs

} patterns
Program Queries

- identify elements implementing
  - high-level design
  - refactoring preconditions
  - run-time bugs

- program queries
  - abstract pattern definition
  - concrete pattern instances

- user-defined

patterns
Program Queries

- User-defined patterns
- Program queries
  - Abstract pattern definition
  - Concrete pattern instances

- Efficacious detection
  + Declarative definition

- Elements implementing
  - High-level design
  - Refactoring preconditions
  - Run-time bugs

Identify patterns
Declarative Program Queries

- declarative problem specification
- what
- how
- problem-solving strategy
logic rules describe ancestor relation

```java
isAncestorOf(?root, ?directSubclass) if
  isSubClassOf(?directSubclass, ?root).

isAncestorOf(?root, ?indirectSubclass) if
  isSubClassOf(?indirectSubclass, ?parent),
  isAncestorOf(?root, ?parent).
```

Declarative Program Queries

declarative problem specification

what

how

problem-solving strategy
Declarative Program Queries

logic rules describe ancestor relation

\[
is\text{AncestorOf}(\texttt{root}, \texttt{directSubclass}) \text{ if } \not\text{isSubClassOf}(\texttt{directSubclass}, \texttt{root}).
\]

\[
is\text{AncestorOf}(\texttt{root}, \texttt{indirectSubclass}) \text{ if } \not\text{isSubClassOf}(\texttt{indirectSubclass}, \texttt{parent}),
is\text{AncestorOf}(\texttt{root}, \texttt{parent}).
\]

logic queries initiate a search for solutions

- if isAncestorOf(java.lang.Object, FooBar)
- if isAncestorOf(\texttt{superclass}, FooBar)
- if isAncestorOf(java.lang.Object, \texttt{subclass})
- if isAncestorOf(\texttt{superclass}, \texttt{subclass})

verify java.lang.Object is ancestor of FooBar
find subclasses of java.lang.Object
The Soul Logic Meta Programming Language

- Pattern definition
  - Logic rules over program representation
- Pattern identification
  - SLD-resolution procedure

**Declarative**
- What versus how

**Powerful**
- Recursion
- Backtracking
- Logic connectives
- Multi-directional predicates
- Pattern-matching
- Non-determinism
- Symbiosis


Soul [2001: wuyts]
The Soul Logic Meta Programming Language

pattern definition
logic rules over program representation

pattern identification
SLD-resolution procedure

Soul [2001:wuyts]
Irish [2004:fabry]

class Bar {
    float x;
    public void foo() {
      x = 42 + 3.14d;
    }
}

&lt;assign('float',
        '=?',
     variable('float', 'x'),
     binaryExp('double',
               '=?',
                literal('int', '42'),
                literal('double', '3.14d')))&gt;

return(?expression)
variable(?type,?identifier)
assign(?type, ?operator, ?lhs, ?rhs)
for(?init, ?expression, ?update, ?body)
send(?type, ?receiver, ?message ?argumentList)
binaryExp(?type, ?operator, ?lOperand, ?rOperand)
...

source code
meta model
class

declarative
✓ what versus how

powerful
✓ recursion
✓ backtracking
✓ logic connectives
✓ multi-directional predicates
✓ pattern-matching
✓ non-determinism
✓ symbiosis

http://prog.vub.ac.be/SOUL/
Detecting Implementation Patterns

class Y {
    private X var;

    public X getVar {
        return var;
    }
}

getterMethod(?class, ?method, ?field) if
    isMethodInClass(?method, ?class),
    fieldInClassChain(?field, ?class),
    fieldName(?field, ?fname),
    methodStatements(?method, ?slist),
    ?slist = <return(variable(ctype, ?fname))>
Detecting Implementation Patterns

```java
class Y {
    private X var;

    public X getVar {
        return var;
    }
}
```

```java
getterMethod(?class, ?method, ?field) if
    isMethodInClass(?method, ?class),
    fieldInClassChain(?field, ?class),
    fieldName(?field, ?fname),
    methodStatements(?method, ?slist),
    ?slist = <return(variable(?utype, ?fname))>
```

**Operational definition of search for pattern variants**

**Declarative specification of pattern's essence**
Detecting Implementation Patterns

```
class Y {
    private X var;
    public X getVar {
        return var;
    }
}
```

`getterMethod(?class, ?method, ?field) if
isMethodInClass(?method, ?class),
fieldInClassChain(?field, ?class),
fieldName(?field, ?fname),
methodStatements(?method, ?slist),
?slist = <return(variable(?vtype, ?fname))>)
```

```
class Y {
    private X var;
    private Y self() {
        return this;
    }
    public X getVar {
        return this.self().var;
    }
}
```

behavioral program representations necessary

various non-trivial candidates

unfamiliar to application programmers
Detecting Implementation Patterns

```java
class Y {
    private X var;
    public X getVar {
        return var;
    }
}
```

getterMethod(?class, ?method, ?field) if
isMethodInClass(?method, ?class),
fieldInClassChain(?field, ?class),
fieldName(?field, ?fname),
methodStatements(?method, ?slist),
?slist = <return(variable(?utype, ?fname)))>

unfamiliar logic program representations

prototypical source code implementation
Extension: Logic Source Code Templates

```java
if jtStatement(?statement){  ?x = (?type) ?e; }
```
Extension: Logic Source Code Templates

```java
if jtStatement(?statement){  ?x = (?type) ?e; }
```

- grammar production rule
- source code with logic variables
- matching base program elements
Extension: Logic Source Code Templates

- a new kind of condition
- multi-directionality
- non-determinism
- unification of variables

```java
if jtStatement(?statement){  ?x = (?type) ?e; }
```

- grammar production rule
- matching base program elements
- source code with logic variables

unification of variables
Soul + Source Code Templates

**extended SOUL**

*declarative pattern specification*

concrete source code of prototypical implementation

*efficacious pattern identification*

resolve templates against **behavioral representations**

employ logics of **quantified truth**
Soul + Source Code Templates

- **declarative pattern specification**
  - concrete source code of prototypical implementation

- **efficacious pattern identification**
  - resolve templates against behavioral representations
  - employ logics of quantified truth

- handle inevitable approximations and unknowns
- quantify similarity of template and its matches
Behavioral Program Representations

run-time values of expressions

```java
public X getVar {
    return this.self().var;
}
```
Behavioral Program Representations

run-time values of expressions

```
public X getVar {
    return this.self().var;
}
```

obtained through static analysis

simulate execution with abstract descriptions of possible values

The underlying abstract-interpretation semantics even odd

Parity \( \gamma : \text{Parity} \rightarrow P(\text{Int}) \)

\( \gamma(\text{even}) = \{\ldots, -2, 0, 2, \ldots\} \)

This semantics ensures that every transition in the original, "concrete" semantics is simulated by one in the abstract semantics.

```
public X getVar {
    return this.self().var;
}
```

```
Parity
   even       odd

parity
```

```
interpreter
abstract interpretation
abstract domain
```

```
Parity
   even       odd

parity
```

Behavioral Program Representations

run-time values of expressions

obtained through static analysis
simulate execution with abstract descriptions of possible values

points-to sets as abstract value descriptions
objects shape overall program behavior
Behavioral Program Representations

run-time values of expressions

```
public x getVar {
    return this.self().var;
}
```

obtained through static analysis

simulate execution with abstract descriptions of possible values

points-to sets as abstract value descriptions

objects shape overall program behavior
Resolving Expression Templates

```
jtExpression(?'cast){ (java.lang.Object) ?expression }
```
Resolving Expression Templates

constraints on possible bindings

```
jtExpression(?cast){ (java.lang.Object) ?expression }
```

1/ `cast` expression in base program
2/ casting to type `java.lang.Object`
3/ compatible subexpression
Resolving Expression Templates

```java
jtExpression(?cast){ (java.lang.Object) ?expression }
```

1/ `cast` expression in `base` program
2/ casting to `type` `java.lang.Object`
3/ `compatible` subexpression

- might `differ` `syntax-wise`
- must have `compatible abstract value descriptions` intersecting points-to sets
Resolving Expression Templates

Constraints on possible bindings

\[
\text{jtExpression(?cast)}\{ \ (\text{java.lang.Object}) \ ?\text{expression} \ }
\]

1/ cast expression in base program
2/ casting to type `java.lang.Object`
3/ compatible subexpression

- might differ syntax-wise
- must have compatible abstract value descriptions
- intersecting points-to sets
- may-alias based unification algorithm

[2006:ppdp:deroover]
if jtStatement(?s1){ return ?expression; },
jtStatement(?s2){ return ?expression; },
differs(?s1, ?s2)
Resolving Statement Templates

```java
if (jtStatement(?s1) { return ?expression; },
    jtStatement(?s2) { return ?expression; },
differs(?s1, ?s2)
```

<table>
<thead>
<tr>
<th>?s2</th>
<th>?s1</th>
</tr>
</thead>
<tbody>
<tr>
<td>return foo;</td>
<td>return foo;</td>
</tr>
<tr>
<td>return this.foo;</td>
<td>return this.foo;</td>
</tr>
<tr>
<td>return this.self().foo;</td>
<td>x = foo; return x;</td>
</tr>
<tr>
<td>return o.returnArgument(foo);</td>
<td>syntactically differing return statements possibly returning overlapping sets of objects</td>
</tr>
</tbody>
</table>
### Resolving Statement Templates

```java
if jtStatement(?s1){ return ?expression; },
jtStatement(?s2){ return ?expression; },
differs(?s1, ?s2)
```

**constraint over values returned at run-time**

**syntactically differing return statements**

possibly returning overlapping sets of objects
Resolving Statement Sequence Templates

```java
public X getVar {
    log("var accessed");
    return var;
}
```

Constraints on possible bindings

```java
if jtStatement(?block){  ?s1;  ?s2;  ?s3;}
```
Resolving Statement Sequence Templates

```java
public X getVar {
    log("var accessed");
    return var;
}
```

constraints on possible bindings

```java
if jtStatement(\?block){ \?s1; \?s2; \?s3;}
```

1/ **containment** constraint
   multiple layers of indirections

2/ **ordering** constraint
   intertwined statements
Resolving Statement Sequence Templates

```java
public X getVar {
    log("var accessed");
    return var;
}
```

constraints on possible bindings

```java
if jtStatement(\?block)\{ \?s1; \?s2; \?s3;\}
```

1/ **containment** constraint
   multiple layers of indirections

2/ **ordering** constraint
   intertwined statements
Detecting “Concurrent Modification” Exceptions

```java
if jtStatement(s) {
    while(iterator.hasNext()) {
        collection.add(element);
    }
},
jtExpression(iterator){collection.iterator()}
```
Detecting “Concurrent Modification” Exceptions

```java
if (jtStatement) {
    while (iterator.hasNext()) {
        collection.add(element);
    }
}

jtExpression(iterator){collection.iterator()}

public List list;

public void insertElement(Object x) {
    Iterator i = list.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        operation(x, (Collection) this.self().list);
    }
}

public void operation(Object o, Collection c) {
    c.add(o);
}
```
Detecting “Concurrent Modification” Exceptions

```java
public void operation(Object o, Collection c) {
    c.add(o);
}
```
Fine-grained Program Queries

efficacious detection + declarative definition
Fine-grained Program Queries

efficacious detection + declarative definition

concrete source code templates

```java
if jtClassDeclaration(?c){
  class ?c {
    private ?type ?field;
    public ?type ?name() { return ?field; }
  }
}
```
Fine-grained Program Queries

efficacious detection + declarative definition

concrete source code templates

```java
if jtClassDeclaration(?c){
    class ?c {
        private ?type ?field;
        public ?type ?name() { return ?field; }
    }
}
```

resolved non-syntactically

unify syntactically ≠ expressions

⇔

compatible abstract value descriptions
Fine-grained Program Queries

concrete source code templates

```java
if jtClassDeclaration(?c){
    class ?c {
        private ?type ?field;
        public ?type ?name() { return ?field; }
    }
}
```

resolved non-syntactically

- unify syntactically ≠ expressions
  - compatible abstract value descriptions

logics of quantified truth

- efficacious detection + declarative definition

[2007:pepm:deroover]

[2006:ipdp:deroover]
Ongoing Work & Collaboration Possibilities

A3: Model Evolution
WP4: Consistency Checking and Coevolution
T2: Using declarative meta-programming techniques

A2: Model Analysis
WP6: Verification Methods and Tools
T2: Model extraction
quantifying flow paths

must template match on one, all or specific paths?

higher-order logic operators needed

A3: Model Evolution
WP4: Consistency Checking and Coevolution
T2: Using declarative meta-programming techniques

A2: Model Analysis
WP6: Verification Methods and Tools
T2: Model extraction
Ongoing Work & Collaboration Possibilities

**quantifying flow paths**

must template match on one, all or specific paths?
higher-order logic operators needed

**demarcating execution contexts**

context-sensitive program analyses
usually only considered for preciseness
want to peruse context information
experience with logics of qualified truth [2006:icpc:deroover]
Ongoing Work & Collaboration Possibilities

quantifying flow paths
must template match on one, all or specific paths?
higher-order logic operators needed

demarcating execution contexts
context-sensitive program analyses
usually only considered for preciseness
want to peruse context information
experience with logics of qualified truth [2006:icpc:deroover]

alternative behavioral program representations
generic abstract domains for OO-programs