Behavioral Program Queries using Logic Source Code Templates

Coen De Roover ▲ Johan Brichau*

▲ Programming Technology Lab - Vrije Universiteit Brussel - Belgium

* Département d’Ingénierie Informatique - Université catholique de Louvain - Belgium
Context: Program Queries

- Implementing elements
- Faulty well-known prescribed suboptimal behavior

Effective identification + declarative specification

Abstract definition → Concrete instances
Context: Program Queries

Implementing elements:
- faulty
- well-known
- prescribed
- suboptimal

Heterogeneous code behavior

Abstract definition + declarative specification = effective identification

Concrete instances
Introduction

Context: Program Queries

- Implementing elements
  - Faulty, well-known, prescribed, suboptimal behavior

- Identification of similar behavior in heterogeneous code

- Abstract definition and declarative specification

- Discrepancies:
  - Users resort to enumeration operational queries

Effective identification + concrete instances
Motivating Example: Detecting Getter Methods

```plaintext
1. ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2. ?f isFieldDeclarationInClassDeclaration: ?c,
3. ?f fieldDeclarationHasFragment: ?g,
4. ?g variableDeclarationFragmentHasName: ?name,
5. ?m isMethodDeclarationInClassDeclaration: ?c
6. ?m methodDeclarationHasBody: block(?s),
7. ?s contains: returnStatement(?name)
```

Should detect getter methods in an imaginary* LMP language.

* any similarities with Soul minus the extensions I’m about to present are purely incidental and unintentional ;(
Motivating Example: Detecting Getter Methods

Should detect getter methods in an imaginary* LMP language.

class X {
private x var;
public x getVar() {
    return var;
}
}

* any similarities with Soul minus the extensions I'm about to present are purely incidental and unintentional ;}
Motivating Example: Detecting Getter Methods

Should detect getter methods in an imaginary LMP language.

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

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

*any similarities with Soul minus the extensions I’m about to present are purely incidental and unintentional ;)*
Motivating Example: Detecting Getter Methods

It should detect getter methods in an imaginary* LMP language.

*any similarities with Soul minus the extensions I’m about to present are purely incidental and unintentional ;)

```java
class Y {
    private X var;
    public X getVar() {
        return var;
    }
}
```
Isn’t there a Straightforward Solution?

query behavior instead of code

effective identification +
declarative specification
Isn’t there a Straightforward Solution?

query behavior instead of code

effective identification + declarative specification

multiple possibilities
Isn’t there a Straightforward Solution?

query behavior instead of code

effective identification +
declarative specification

unfamiliar

multiple possibilities

r0 := @this: testapp2.Example;
r1 = r0.<testapp2.Example: java.lang.Integer buffer>;
r0.<testapp2.Example: java.lang.Integer buffer> = null;
return r1;
Isn’t there a Straightforward Solution?

query behavior instead of code

effective identification + declarative specification

unfamiliar

multiple possibilities

handle with care

r0 := @this: testapp2.Example;
r1 = r0.<testapp2.Example: java.lang.Integer buffer>;
r0.<testapp2.Example: java.lang.Integer buffer> = null;
return r1;
So, instead ..

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

public Integer gethour() {
    return this.hour;
}
public Integer gethourlazy() {
    if(hour==null) {
        hour = this.currentHour();
        return hour;
    }
}
public Integer getBuffer() {
    Integer temp;
    temp = this.self().buffer;
    return temp;
}
```
Solution Cornerstones

- **declarative pattern specification**
  - concrete source code templates
  - integrated in logic language

- **effective pattern identification**
  - resolved by fuzzy logic program
  - behavioral similarity determined statically
  - by open unification framework
Solution Cornerstones

- concrete source code templates integrated in logic language
  - user familiarity
  - declarative pattern specification

- effective pattern identification
  - resolved by fuzzy logic program
  - behavioral similarity determined statically
  - by open unification framework
Solution Cornerstones

- Concrete source code templates integrated in logic language
- Template composition
- User familiarity
- Declarative pattern specification
- Effective pattern identification

Resolved by fuzzy logic program
Behavioral similarity determined statically by open unification framework
Solution Cornerstones

- **user familiarity**
- **concrete source code templates**
- **declarative pattern specification**
- **integrated in logic language**
- **template composition**
- **uniform unification** across templates and ordinary conditions

**effective pattern identification**

- **resolved by fuzzy logic program**
- **behavioral similarity** determined statically
- **by open unification framework**
Solution Cornerstones

user familiarity

**concrete source code templates**

integrated in logic language

uniform **unification**

across templates and ordinary conditions

**declarative pattern specification**

**template composition**

**effective pattern identification**

similar template/match

resolved by **fuzzy** logic program

behavioral similarity determined statically

by open **unification** framework
Solution Cornerstones

- **User familiarity**
  - Concrete source code templates
  - Integrated in logic language
  - Uniform unification across templates and ordinary conditions

- **Declarative pattern specification**
  - Template composition

- **Effective pattern identification**
  - Similarity template/match
  - Resolved by fuzzy logic program
  - Customizable
  - Behavioral similarity determined statically by open unification framework
Solution Cornerstones

User familiarity

Concrete source code templates

Declarative pattern specification

Integrated in logic language

Template composition

Uniform unification across templates and ordinary conditions

Effective pattern identification

Similarity template/match

Resolved by fuzzy logic program

Behavioral similarity determined statically

By open unification framework

Transparent to user

Customizable
Solution Cornerstones

user familiarity

**concrete source code templates**

**declarative pattern specification**

**template composition**

integrated in logic language

uniform **unification** across templates and ordinary conditions

**effective pattern identification**

similarity template/match

**behavioral similarity** determined statically by open **unification framework**

resolved by **fuzzy logic program**

customizable

family of analyses

transparant to user
Solution Cornerstones

user familiarity

congrue source code templates

declarative pattern specification

integrated in logic language

template composition

uniform unification across templates and ordinary conditions

effective pattern identification

similarity template/match

resolved by fuzzy logic program

customizable

behavioral similarity determined statically

by open unification framework

approximated results

family of analyses

transparent to user
Fuzzy Logic Programming

logic of quantified truth
rules annotated with partial truth degrees
✓ confidence in instances detected by rule

weighted logic rules

\[ q : c \text{ if } q_1, \ldots, q_n \text{ where } c \in ]0,1] \]

similar to f-Prolog
[1990:liu]
Fuzzy Logic Programming

logic of quantified truth
rules annotated with partial truth degrees
✓ confidence in instances detected by rule

weighted logic rules

\[ q : c \text{ if } q_1, \ldots, q_n \text{ where } c \in [0,1] \]

confidence in conclusion \( q \) given absolute truth of \( q_1, \ldots, q_n \)

similar to f-Prolog
[1990:liu]
Fuzzy Logic Programming

logic of quantified truth
rules annotated with partial truth degrees
✓ confidence in instances detected by rule

fuzzy resolution procedure
conclusions from partially satisfied premises
✓ detect partially adhering instances

weighted logic rules
\[ q : c \text{ if } q_1, \ldots, q_n \quad \text{where } c \in ]0,1] \]

fuzzy resolution procedure
\[ T(q) = c \times \min(T(q_1), \ldots, T(q_n)) \]

confidence in conclusion \( q \) given absolute truth of \( q_1, \ldots, q_n \)

similar to f-Prolog [1990:liu]
Fuzzy Logic Programming

logic of quantified truth
rules annotated with partial truth degrees
✓ confidence in instances detected by rule

fuzzy resolution procedure
conclusions from partially satisfied premises
✓ detect partially adhering instances

weighted logic rules
$q : c \text{ if } q_1, \ldots, q_n$ where $c \in ]0,1]$

fuzzy resolution procedure
$\tau(q) = c \times \min(\tau(q_1), \ldots, \tau(q_n))$

confidence in conclusion $q$ given absolute truth of $q_1, \ldots, q_n$

similar to f-Prolog [1990:liu]
Fuzzy Logic Programming

weighted logic rules

\[ q : c \text{ if } q_1, \ldots, q_n \text{ where } c \in [0,1] \]

fuzzy resolution procedure

\[ \tau(q) = c \times \min(\tau(q_1), \ldots, \tau(q_n)) \]

similar to f-Prolog [1990:liu]

confidence in conclusion \( q \) given absolute truth of \( q_1, \ldots, q_n \)
Fuzzy Logic Programming

similar to f-Prolog [1990:liu]

\[ \tau(q) = c \times \min(\tau(q_1), \ldots, \tau(q_n)) \]

weighted logic rules

\[ q : c \text{ if } q_1, \ldots, q_n \]

where \( c \in [0,1] \)

confidence in conclusion \( q \) given absolute truth of \( q_1, \ldots, q_n \)

confidence in instances detected by rule

fuzzy logic

Solution Cornerstones
Fuzzy Logic Programming

Fuzzy Logic Programming is similar to f-Prolog [1990:liu].

The fuzzy resolution procedure detects partially adhering instances from partially satisfied premises.

\[ \tau(q) = c \cdot \min(\tau(q_1), \ldots, \tau(q_n)) \]

Confidence in conclusion \( q \) given absolute truth of \( q_1, \ldots, q_n \) is calculated using weighted logic rules:

\[ q : c \text{ if } q_1, \ldots, q_n \]

where \( c \in [0, 1] \) represents the confidence in instances detected by rule.

Fuzzy logic rules annotated with partial truth degrees provide a basis for confidence in instances detected.
Open Template Compilation

```java
tjExpression(\texttt{?cast})\{ \texttt{(java.lang.Object)} \texttt{?expression} \}
```
Open Template Compilation

constraints on possible bindings

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

1/ cast expression in base program
2/ unifying types
3/ unifying subexpressions
Open Template Compilation

constraints on possible bindings

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

1/ cast expression in base program
2/ unifying types
3/ unifying subexpressions

compiled by fuzzy logic program called by interpreter

results in multiple logic queries for each parse tree of a template
Open Template Compilation

variants on possible bindings

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

1. cast expression in base program
2. unifying types
3. unifying subexpressions

compiled by fuzzy logic program called by interpreter
results in multiple logic queries for each parse tree of a template

\[ \text{if \ jtStatement(?block)\{ ?s1; ?s2; ?s3;\}} \]

weighted by their tolerance for mismatches

layers of indirection, intertwining statement, loops, ...
Multiple Program Representations
Multiple Program Representations

Abstract Syntax Tree

live org.eclipse.jdt.core.dom.ASTNode instances

NOT reified as logic facts
Multiple Program Representations

Abstract Syntax Tree

live org.eclipse.jdt.core.dom.ASTNode instances
Multiple Program Representations

Abstract Syntax Tree

live org.eclipse.jdt.core.dom.ASTNode instances

NOT reified as logic facts
Multiple Program Representations

Abstract Syntax Tree

live `org.eclipse.jdt.core.dom.ASTNode` instances

NOT reified as logic facts

Dataflow Analysis: points-to

safe approximation of run-time objects a reference points to

```
X.y → { AllocNode 3 new java.lang.Integer in method
  <testapp2.Example: void main(java.lang.String[])>,
  ...
  AllocNode 7 new java.lang.Integer in method
  <testapp2.Example: void main(java.lang.String[])> }
```
Multiple Program Representations

Abstract Syntax Tree
live org.eclipse.jdt.core.dom.ASTNode instances
NOT reified as logic facts

Dataflow Analysis: points-to
safe approximation of run-time objects a reference points to

\[
X \rightarrow \begin{cases} \text{new java.lang.Integer} & \text{in method \<testapp2.Example: void main(java.lang.String[])>}, \\
\text{\ldots} \\
\text{new java.lang.Integer} & \text{in method \<testapp2.Example: void main(java.lang.String[])>} \end{cases}
\]

Control flow between within methods
Multiple Program Representations

9

Abstract Syntax Tree

NOT reified as logic facts

org.eclipse.jdt.core.dom.ASTNode instances

Dataflow Analysis: points-to

safe approximation of run-time objects

A reference points to

{ AllocNode 3 new java.lang.Integer in method <testapp2.Example: void main(java.lang.String[])>, ...

AllocNode 7 new java.lang.Integer in method <testapp2.Example: void main(java.lang.String[])> }

Control flow between within methods

Solution Cornerstones
Multiple Program Representations

Abstract Syntax Tree

live `org.eclipse.jdt.core.dom.ASTNode` instances

NOT reified as logic facts

Dataflow Analysis: points-to

safe approximation of run-time objects a reference points to

X.y → { AllocNode 3 new `java.lang.Integer` in method <testapp2.Example: void main(java.lang.String[])>,
... AllocNode 7 new `java.lang.Integer` in method <testapp2.Example: void main(java.lang.String[])> }
Unification with logic terms

Unification with other instances
Unification with logic terms

easily identify elements of interest

```smalltalk
if ?c isCompilationUnit,
   [ ?c types size > 1]

if compilationUnit(packageDeclaration(simpleName(['testapp']), ?, ?) isCompilationUnit

if ?c isCompilationUnit,
   ?c hasPackage: ?p,
   ?p hasName: ?n,
   ?n isSimpleName,
   ?n hasIdentifier: ['testapp']
```

Unification with other instances
Unification with logic terms  

easily identify elements of interest

```prolog
1  if ?c isCompilationUnit,
2     [?c types size > 1]

3  if compilationUnit(packageDeclaration(simpleName([‘testapp’])), ?, ?) isCompilationUnit

4  if ?c isCompilationUnit,
5     ?c hasPackage: ?p,
6     ?p hasName: ?n,
7     ?n isSimpleName,
8     ?n hasIdentifier: [‘testapp’]
```

Unification with other instances
**Smalltalk Open Unification Language**

**Unification with logic terms**  
powered by structural reflection

- easily identify elements of interest

```plaintext
if ?c isCompilationUnit, 
   [?c types size > 1]

if compilationUnit(packageDeclaration(simpleName(['testapp'])), ?, ?) isCompilationUnit

if ?c isCompilationUnit, 
   ?c hasPackage: ?p, 
   ?p hasName: ?n, 
   ?n isSimpleName, 
   ?n hasIdentifier: ['testapp']
```

**Unification with other instances**  
overrides default identity comparison

- implement recurring comparisons

Expression  \( \cong \)  ParenthesizedExpression

Type  \( \cong \)  TypeDeclaration

SimpleName  \( \cong \)  QualifiedName
**Smalltalk Open Unification Language**

**Unification with logic terms** powered by structural reflection

easily identify elements of interest

```
1 if ?c isCompilationUnit,
2   [ ?c types size > 1]

3 if compilationUnit(packageDeclaration(simpleName(['testapp']), ?, ?) isCompilationUnit

4 if ?c isCompilationUnit,
5   ?c hasPackage: ?p,
6   ?p hasName: ?n,
7   ?n isSimpleName,
8   ?n hasIdentifier: ['testapp']
```

**Unification with other instances** overrides default identity comparison

implement recurring comparisons

Expression 2 ParenthesizedExpression

Type 2 TypeDeclaration

SimpleName 2 QualifiedName
**Smalltalk Open Unification Language**

**Unification with logic terms** powered by structural reflection

easily identify elements of interest

```plaintext
1 if ?c isCompilationUnit,
2    [ ?c types size > 1]

3 if compilationUnit(packageDeclaration(simpleName(['testapp']), ?, ?) isCompilationUnit

4 if ?c isCompilationUnit,
5    ?c hasPackage: ?p,
6    ?p hasName: ?n,
7    ?n isSimpleName,
8    ?n hasIdentifier: ['testapp']
```

**Unification with other instances** overrides default identity comparison

implement recurring comparisons

Expression = ParenthesizedExpression
Type... = TypeDeclaration
SimpleName = QualifiedName

**Solution Cornerstones - Open Unification**
Unification based on Semantic Analysis

across logic and template conditions
Unification based on Semantic Analysis

Conciseness across logic and template conditions
Unification based on Semantic Analysis

Correctness

Conciseness

across logic and template conditions
Unification based on Semantic Analysis

name resolution

1 if jtClassDeclaration(?class)
2    class FooClass extends testapp2.Example implements Collection {}
3 }
4 if jtClassDeclaration(?class)
5    class FooClass extends Example implements java.util.Collection {}
6 }

package testapp2;
public class FooClass extends testapp2.Example implements Collection {}
Unification based on Semantic Analysis

name resolution

```java
1 if jtClassDeclaration(?class) {
2     class FooClass extends testapp2.Example implements Collection {}
3 }
4 if jtClassDeclaration(?class) {
5     class FooClass extends Example implements java.util.Collection {}
6 }
```

package testapp2;
public class FooClass extends testapp2.Example implements Collection {}

scoping rules

```prolog
1 ?m getsFragment: ?g offFieldDeclaration: ?f in: ?c if
2 ?f isFieldDeclarationInClassDeclaration: ?c, 
3 ?f fieldDeclarationHasFragment: ?g, 
4 ?g variableDeclarationFragmentHasName: ?name, 
5 ?m isMethodDeclarationInClassDeclaration: ?c 
6 ?m methodDeclarationHasBody: block(?s), 
7 ?s contains: returnStatement(?name)
```

private Integer f;
public Integer notGettingF(Integer f) { return f; }

Conciseness

Correctness

across logic and template conditions
Unification based on Points-to Analysis

```java
if (jtStatement(\$s1)) { return \$expression; },
jtStatement(\$s2){ return \$expression; },
differs(\$s1, \$s2)
```
Unification based on Points-to Analysis

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

Syntactically differing return statements possibly returning overlapping sets of objects
Unification based on Points-to Analysis

\[\text{if } \text{jtStatement}(?s1)\{ \text{return } ?\text{expression}; \}, \]
\[\text{jtStatement}(?s2)\{ \text{return } ?\text{expression}; \}, \]
\[\text{differs}(?s1, ?s2)\]

**Behavioral Similarity**

<table>
<thead>
<tr>
<th>?s1</th>
<th>?s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1 = return foo;</td>
<td>return foo;</td>
</tr>
<tr>
<td>value2 = return this.foo;</td>
<td>return this.foo;</td>
</tr>
<tr>
<td>value3 = return this.self().foo;</td>
<td>return this.self().foo;</td>
</tr>
<tr>
<td>x = foo; return x;</td>
<td>x = foo; return x;</td>
</tr>
<tr>
<td>return o.returnValueArgument(foo);</td>
<td>return o.returnValueArgument(foo);</td>
</tr>
</tbody>
</table>

constraint over values returned at run-time

syntactically differing return statements possibly returning overlapping sets of objects
Template Resolution Semantics

design guideline

close to concrete source code of prototypical implementation
match many implementation variants

interpreted as freely as possible

**BUT** every single element in template adds constraints

```
while(?cond) ?s;
vs
while(?cond) { ?s; }
```

HOWEVER: an open implementation, so amenable by application programmer
Template Semantics: Sequences

```plaintext
{ 
  a(); 
  { ?s2; };
  b();
}

a() {
  ?s1;
}

?s1

b() {
  c();
}

2

b() {
  ?s2;
}

2.1

c() {
  ?s3;
}
```

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

1/ **control flow** constraint

- `?s`'s in interprocedural control-flow of `?block`  
- `?s2` follows: `?s1`, `?s3` follows: `?s2`

2/ **kind** constraint (statement or expression statement)
Template Semantics: Statements and Expressions

```
public A method() {
    A local = ?x.m();
    return local;
}
```

- `jtMethodDeclaration(?m) {}` has a **return statement** `return(?val)` among its **lexical** statements.
- `?modList ?type ?name(?argList) {} return ?x.m();` has an expression `?e` which matches `?x.m()` in its inter-procedural **control flow** (before the statement `statement`).
- The statement’s returned expression `?val` **unifies with** `?e`.
An Example: Concurrent Modification Exceptions

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

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);
}
```
An Example: Concurrent Modification Exceptions

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

```java
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);
}
```
Summary

concrete code templates integrated in logic program query language
  unification uniform across regular and template conditions
  composition by logic connectives
  open template compilation rules

quantified similarity template-match
  fuzzy logic

open unification framework
  incorporates static analyses in user-transparent way
    query conciseness & correctness
    statically determined behavioral similarity
The End