Towards Abstract Interpretation for Recovering Design Information

Coen De Roover, Kris Gybels, Theo D’Hondt

Programming Technology Lab
Vrije Universiteit Brussel
Belgium
program maintenance is hard

requires understanding of legacy code

BUT
design information is often not up-to-date or missing

reverse engineering techniques
recover design information from source code

HOWEVER

pattern detection

• detection based on structural relations
• no formal model for detection based on run-time behaviour
Logic Meta Programming

uses a logic language for reasoning about programs

- architectural model checking
- detect and enforce software patterns
- support software evolution

Object-Oriented Program

represented

Meta Model

Element isClass
Element hasMethod: m
...

reason

Logic Rules & Queries

if ?x isInHierarchyOf: Element
A Logic Meta Programming Example

```prolog
?directSubclass isInHierarchyOf: ?root if

?indirectSubclass isInHierarchyOf: ?root if
  ?indirectSubclass isSubclassOf: ?parent, ?parent isInHierarchyOf: ?root
```

- the `isInHierarchyOf` predicate can both
  - verify the existence of a hierarchy relation
  - detect all classes in hierarchy of another class
Software Patterns

- generalisations of commonly recurring programming constructs in non-arbitrary contexts
  - language specific idioms
  - anti-patterns or bad smells
  - architectural design patterns
    - structural perspective
    - behavioural perspective
  - can be detected to recover design information
  - running example: the Visitor design pattern
Visitor Through Structural Perspective

![Diagram of Visitor Design Pattern]

- **Element**
  - accept: visitor

- **ConcreteElementA**
  - someMessage
  - accept: visitor
  - visitor visitConcreteElementA: self

- **ConcreteElementB**
  - someMessage
  - accept: visitor
  - visitor visitConcreteElementB: self

- **Visitor**
  - visitConcreteElementA:
  - visitConcreteElementB:

- **ConcreteVisitor**
  - visitConcreteElementA:
  - visitConcreteElementB:
Visitor Through Behavioural Perspective

Fig. 3. An annotated sequence diagram demonstrating the recursive nature of the Visitor design pattern.

1. ?visitor with selector: ?visitSelector
2. visits: ?element
   with selector: ?accept
   if
   3. ?visitor isClass,
   4. ?element isClass,
   5. ?element implements: ?accept
   with body: ?acceptBody,
   6. ?acceptBody method arguments: ?acceptArgs,
   7. ?acceptBody method statements:
      <return(send(?visitor, ?visitSelector, ?visitArgs))>,
   8. ?visitArgs contains: variable([#self]),

Fig. 4. Structural Visitor decision rule

   invokedBy: ?invoker
   if
   2. ?invoker double dispatches on: ?composite
   3. selector: ?acceptSelector
   at: ?begin
   and on: ?visitor
   selector: ?visitSelector
   at: ?end,

Fig. 5. Behavioural Visitor detection rule

1. method with selector.
2. There must also be a class bound to the element variable which must implement a method name ?accept implemented as ?accept Body.
3. The Visitor protocol demands that this method is called with the visiting object as its argument which is stated on the last line.
Detection of the Visitor Structure

```prolog
  ?visitor isClass,
  ?visitor implements: ?visitSelector,
  ?element isClass,
  ?element implements: ?accept withBody: ?acceptBody,
  ?acceptBody methodArguments: ?acceptArgs,
  ?acceptBody methodStatements:
    <return(send(?visitor, ?visitSelector, ?visitArgs ))>,
  ?visitArgs contains: variable([#self]),
```
This rule fails horribly

many implementations of behaviour possible

- alternative naming scheme
- indirect argument passing
- control over traversal
- different visited structures
- ...

we want one rule to detect them all
Limitations of the structural meta model

- strongly tied to source code
- queries are expressed as rigid conditions over structural elements
- hard to express highly dynamic patterns
- rather language dependent
- too brittle for use in real-life situations
A new behavioural meta model

Object-Oriented Program

Meta Model

Logic Rules & Queries

methodEntry(?sequenceNumber, ?sendingInstance, ?receivingInstance, ?receivedSelector, ?receivedArguments)

methodExit(?sequenceNumber, ?methodInvocationNumber, ?returnedValue)

methodEntry(21, 4, 6, accept:, [9])
    methodEntry(22, 6, 7, accept', [9])
    methodEntry(23, 7, 9, visitParagraph:, [7])
    methodExit(24, 23, 10)

assignment(?sequenceNumber, ?methodInvocationNumber, ?instance, ?variable, ?value)

recorded as

Figure 4.2: UML diagram of the run-time events class... above predicates. We could do the same with the post-mortem analysis variant, but the equivalent transcription to Prolog...
An example execution trace

**composite instance**

```
Chapter
  section
    accept: aVisitor
      ^aVisitor visitChapter: self
```

**source code**

```
Chapter>>accept: aVisitor
  section accept: aVisitor
    ^aVisitor visitChapter: self
```

**source code**

```
Section>>accept: aVisitor
  paragraphs do:
    [:par | par accept: aVisitor]
    ^aVisitor visitSection: self
```

**source code**

```
Paragraph>>accept: aVisitor
  ^aVisitor visitParagraph: self
```

**example trace: “aSection accept: aChapterVisitor”**

```
methodEntry(21,4,6,’#accept:’,[9]).
  methodEntry(22,6,7,’#accept:’,[9]).
    methodEntry(23,7,9,’#visitParagraph:’,[7])
    methodExit(24,23,10).
  methodExit(25,22,10).
methodEntry(26,6,9,’#visitSection:’,[6]).
  methodExit(27,26,10).
methodExit(28,21,10).
```

---

**Diagram: An example execution trace**

- **Composite instance**
  - `Chapter`
    - `section`
      - `accept: aVisitor`
        - `^aVisitor visitChapter: self`

- **Source code**
  - `Chapter>>accept: aVisitor`
    - `section accept: aVisitor`
      - `^aVisitor visitChapter: self`
  - `Section>>accept: aVisitor`
    - `paragraphs do:
          [:par | par accept: aVisitor]
          ^aVisitor visitSection: self`
  - `Paragraph>>accept: aVisitor`
    - `^aVisitor visitParagraph: self`

**Example trace:**
```
methodEntry(21,4,6,’#accept:’,[9]).
  methodEntry(22,6,7,’#accept:’,[9]).
    methodEntry(23,7,9,’#visitParagraph:’,[7])
    methodExit(24,23,10).
  methodExit(25,22,10).
methodEntry(26,6,9,’#visitSection:’,[6]).
  methodExit(27,26,10).
methodExit(28,21,10).
```
Detection of the Visitor behaviour

straightforward translation of the Visitor behaviour

insensitive to implementation differences

```prolog
?-invoker doubleDispatchesOn: ?composite
   selector: ?acceptselector
   at: ?begin
?-invoker doubleDispatchesOn: ?composite
   selector: ?acceptselector
   at: ?begin
?-invoker doubleDispatchesOn: ?composite
   selector: ?acceptselector
   at: ?begin
?-invoker doubleDispatchesOn: ?composite
   selector: ?acceptselector
   at: ?begin

forall: (?composite contains: ?part at: ?begin)
```
Results

Composite instance | Source code
--- | ---
aChapter | Chapter>>accept: aVisitor
| section accept: aVisitor
| ^aVisitor visitChapter: self

Source code

example trace: "aSection accept: aChapterVisitor"

```
methodEntry(21,4,6,'#accept:',[9]).
methodEntry(22,6,7,'#accept:',[9]).
methodEntry(23,7,9,'#visitParagraph:',[7])
methodExit(24,23,10).
methodExit(25,22,10).
methodEntry(26,6,9,'#visitSection:',[6]).
methodExit(27,26,10).
methodExit(28,21,10).
```


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a Chapter</td>
<td>a VisitorInvoker</td>
<td>20</td>
<td>30</td>
<td>a ChapterVisitor</td>
</tr>
<tr>
<td>a Section</td>
<td>a Chapter</td>
<td>21</td>
<td>27</td>
<td>a ChapterVisitor</td>
</tr>
<tr>
<td>a Paragraph</td>
<td>a Section</td>
<td>22</td>
<td>24</td>
<td>a ChapterVisitor</td>
</tr>
</tbody>
</table>
Towards a semantics-based meta model

behavioural meta model >>> structural meta model

more powerful
more flexible
more expressive

BUT execution traces are only valid for one of many possible program execution paths

BUT ad-hoc approach instead of formal model

use abstract interpretation for a semantics-based meta model
Discussion

- We are new to abstract interpretation
- All pointers welcome!
- Better to keep AI process and logic reasoning process separate or interleave them?
- AI gathers basic program facts
- Should logic rules specialise an abstract interpreter for their purpose?
- New opportunities for LMP
- Proof that a pattern is always present
- Determine the conditions under which a pattern is present
THE END
Accessor method pattern

accessor(?selector,?instance,?variable) if
methodEntry(?sn,?,?receiver,?selector,?),
equals(?selector,?variable),
?vARIABLE in: ?receiver at: ?sn value: ?value,
methodExit(?,,?sn,?value).
Double Dispatching Definition

?invoker doubleDispatchesOn: ?primary
  selector: ?primarySelector
  at: ?primarySN
  andOn: ?secondary
  selector: ?secondarySelector
  at: ?secondaryExit if

methodEntry(?primarySN, ?invoker, ?primary, ?primarySelector, ?primArguments),
methodExit(?primaryExit, ?primarySN, ?),
member(?secondary, ?primArguments),
methodExit(?secondaryExit, ?secondarySN, ?),
member(?primary, ?secArguments),
greater(?primaryExit, ?secondaryExit)
not(?variable isInstVarOf: ?instance),
sequenceTotal(?max),
constrain(?methodInvocationNumber,[1 to: ?max]),
?a assignmentDuring: ?methodInvocationNumber ,
not(? overridesInternalAssignment: ?a with: ?),
equals(?a,assignment(?,?methodInvocationNumber,?instance,?variable,?value))

?variable isInstVarOf: ?instance,
sequenceTotal(?max),
constrain(?methodInvocationNumber,[1 to: ?max]),
?a assignmentPrecedes: ?methodInvocationNumber,
not(? overridesInternalAssignment: ?a with: ?),
equals(?a,assignment(?,?,?instance,?variable,?value))
OverridesExternalAssignment

equals(?a,assignment(?sn,?mn,?instance,?variable,?)),
equals(?x,assignment(?sn2,?mn2,?instance,?variable,?v)),
not(equals(?mn2,?mn)),
greater(?sn2,?sn),
smallerOrEqual(?mn2,?endMN)
Figure 4.1: The results show when exactly each class is instantiated.

It states that an instance of a class is created...