Efficient Identification of Design Patterns with Bit-vector Algorithm

Olivier Kaczor, Yann-Gaël Guéhéneuc and Sylvie Hamel
{kaczorol, guehene, hamelsyl}@iro.umontreal.ca
Plan

1. Introduction
2. Related Work
3. Our Approach:
   1. Code source transformation
   2. Identification with Bit-vector algorithm
4. Case Study
5. Current and Future Work
1. Introduction

Design patterns:

- provide good solutions to recurring design problems
- provide design vocabulary to improve communication
- encourage designs re-use
- auto document the source code

“Design patterns help a designer get a design "right" faster”

[Gamma and al. 1994]
1. Introduction

Why detect design patterns occurrences:

- Measure the quality of a system
- Help document the source code
- Help understand the design problems and choices

Sub-problem: Identify micro-architectures similar to design motifs

- Program model rarely reflects a design motif completely
- Help to improve the code by applying corrections based on the design motifs
1. Introduction

Design patterns detection system:

- Source Code
- Design Patterns Detection
- Results

- Source Code Transformation
- Detection Algorithm
2. Related Work

Some known identification techniques:

- Logic programming [Kramer-Prechelt 1996, Wuyts 1998]
- Constraints programming with explanations [Guéhéneuc-Jussien 2001]
- Fuzzy logic [Jahnke and al. 1997, Niere and al. 2001]
- Graphs transformation [Smolarova-Kadlec 2000]

General problem: limited performance
2. Related Work

Metrics to improve efficiency [Antoniol and al. 1998, Guéhéneuc and al. 2004]

Idea: Reduce the search space by removing entities which obviously do not participate in a design motif according to expected metrics values

1. Create fingerprints of design patterns roles with internal attributes of classes playing those roles
2. Use those fingerprints to create identification rules

Multi-stage-filtering:

\[ D \leq C \leq B \leq A \]
2. Related Work

- Similar problem in bioinformatics:
  - Localising mutated (or not) genes in long anonymous DNA sequences
  - Localising modified (or not) proteins in long amino-acid sequences

- Solutions adopted in bioinformatics:
  - Vectorial algorithms [Myers97, Bergeron-Hamel 2002]
  - Automata simulation [Holub97]
  - Dynamic programming alignment [Needleman-Wunsch70, Smith-Waterman81]
Problem:
- Bioinformatics algorithms work on strings
- Program and design motifs models must be converted in strings

Solution:
- Program and design motifs models can be seen as directed graphs
- Build their string representations by going through every edge in their graphs
Is it possible to find a circuit which uses each edge exactly once?

- To contain such a circuit, the graph must be Eulerian
- A directed graph is Eulerian if and only if every vertex has an equal in-degree and out-degree

We must transform the design motif and program graphs into Eulerian graphs
3. Our approach: Source code transformation (3/4)

- **Steps to obtain the string representations:**

1. Transform the graphs into Eulerian graphs:
   - Build an adjacency matrix to identify vertices with unequal in-degree and out-degree
   - Compute an optimal list of flows to be added between those vertices with the transportation simplex

<table>
<thead>
<tr>
<th>out</th>
<th>Component</th>
<th>Leaf</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Leaf: out-degree: 0
in-degree: 1
3. Our approach: Source code transformation

- **Steps to obtain the string representations:**

2. Find an Eulerian circuit and build the strings:
   - Resolve the directed Chinese Postman Problem to find the shortest tour of the graphs (an Eulerian circuit)
   - Build the strings by running through the circuits

Component in Leaf dm Component in Composite co Component
3. Our approach: Identification algorithm

**Objective:**

Problem: The design motif string is more like a regular expression than a word

Solution: Iterative bit-vector algorithm
3. Our approach: Identification algorithm

Let \( x = x_1x_2...x_m \) be a string representing a program model and \( c \) a symbol in that string.

We define the characteristic vector of \( c \) associated to the string \( x \) to be:

\[
c_i = \begin{cases} 
1 & \text{if } x_i = c \\
0 & \text{otherwise} 
\end{cases}
\]

Characteristic vectors are circular sequences of bits on which we operate with standard bit operations:
- and
- or
- left shift
- right shift
- etc.
3. Our approach: Identification algorithm

Example:

![Diagram]

A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A

\[ G = \underbrace{00000000000000010001000100010000}_{14} \]

\[ in = \underbrace{01010001000100000000000000000000000000}_{19} \]
3. Our approach: Identification algorithm

We match against each other the two string to build a list of potential occurrences.

The algorithm reads triplets of tokens in the design motif string and associates program entities to the roles.

For example:

\[ \text{Component in Leaf} \quad \text{dm} \quad \text{Component in Composite} \quad \text{co Component} \]

We retrieve entities before and after an \textit{in} token in the program string with bit-wise operations on the characteristic vectors to have potential entities for the \textit{Component} and \textit{Leaf} roles.
3. Our approach: Identification algorithm

We search for entities after a Component entity and an in relationship (ex: B in Composite)

\[
\rightarrow B = \overbrace{00001000100010}\ldots 0_{18}
\]
\[
\rightarrow \text{in} = \overbrace{00101000100010}\ldots 0_{18}
\]
\[
(\rightarrow B) \land (\rightarrow \text{in}) = \underbrace{00001000100010}\ldots 0_{18}
\]
\[
E = \underbrace{000000010\ldots 01000000}_{8\ldots 15\ldots 8}
\]
\[
(\rightarrow B) \land (\rightarrow \text{in}) \land E = \underbrace{0000000010}\ldots 0_{8\ldots 22}
\]

<table>
<thead>
<tr>
<th>Component</th>
<th>Leaf</th>
<th>Component</th>
<th>Leaf</th>
<th>Composite</th>
<th>Component</th>
<th>Leaf</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>B</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>B</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

Occurrences after processing the first third and fourth triplets
3. Our approach: Identification algorithm

A program model rarely reflects a design motif completely

- We include automatic and manual approximation mechanisms

1. Association relationships:
   - Composition, aggregation, association and use relationship
   - Conjunction of characteristic vectors

2. Entity counts
   - Ex: A Composite without leaf
   - Triplets can be overlooked

3. Inheritance relationship:
   - Some entities could be inserted or removed from a hierarchy
   - Parent and children of a class are also possible entities for a role
4. Case Study

- We applied our algorithm on 3 public domain software:
  1. Juzzle v0.5
  2. JHotDraw v5.1
  3. QuickUML 2001

- We used an AMD Athlon at 2Ghz with ?? Mb RAM

<table>
<thead>
<tr>
<th>Programs</th>
<th>Sizes</th>
<th>Computation times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Juzzle v0.5</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>JHotDraw v5.1</td>
<td>261</td>
<td>37</td>
</tr>
<tr>
<td>QuickUML 2001</td>
<td>373</td>
<td>149</td>
</tr>
</tbody>
</table>

Computation times (in seconds) for building the string representation
4. Case Study

We compared our bit-vector technique with Ptidej, a framework implementing:

- explanation-based constraint programming
- metric-enhanced explanation-based constraint programming

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>CP+M</th>
<th>BV</th>
<th>BV+O</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract Factory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JHotDraw v5.1</td>
<td>1202</td>
<td>275</td>
<td>218</td>
<td>27</td>
</tr>
<tr>
<td>Juzzle v0.5</td>
<td>1</td>
<td>2</td>
<td>0.9</td>
<td>0.3</td>
</tr>
<tr>
<td>QuickUML 2001</td>
<td>785</td>
<td>153</td>
<td>97</td>
<td>29</td>
</tr>
<tr>
<td><strong>Composite</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JHotDraw v5.1</td>
<td>+∞</td>
<td>17362</td>
<td>129</td>
<td>25</td>
</tr>
<tr>
<td>Juzzle v0.5</td>
<td>45</td>
<td>3</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td>QuickUML 2001</td>
<td>+∞</td>
<td>26514</td>
<td>185</td>
<td>27</td>
</tr>
</tbody>
</table>

Identification times (in seconds) of design motifs with all approximations
4. Case Study

- What is a design motif occurrence?

To compare our results with those of Ptidej, an occurrence is a micro-architecture where each role of the design motif is played by only one entity.

- A Composite design motif with 3 leaves counts as 3 occurrences

<table>
<thead>
<tr>
<th></th>
<th>Existing Occurrences</th>
<th>Exact</th>
<th>Approximate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CP</td>
<td>CP+M</td>
</tr>
<tr>
<td><strong>Abstract Factory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JHotDraw v5.1</td>
<td>0</td>
<td>216/221</td>
<td>104/69</td>
</tr>
<tr>
<td>Juzzle v0.5</td>
<td>0</td>
<td>19/0</td>
<td>0/0</td>
</tr>
<tr>
<td><strong>Composite</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JHotDraw v5.1</td>
<td>70</td>
<td>N/A</td>
<td>0/0</td>
</tr>
<tr>
<td>Juzzle v0.5</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>QuickUML 2001</td>
<td>22</td>
<td>N/A</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Number of identified occurrences of the design motifs with and without ghost entities.
5. Current and Future Work

- Improve the precision of the results
  - Combine bit-vector algorithm with empirical metrics values
  - Add *negative* relationships in design motifs
  - Add dynamic information in the string representations

- Try the approach on more and bigger software

- Try other bioinformatics string matching algorithms