Coordinated Distributed Diagram Transformation for Software Evolution

P. Bottoni, F. Parisi Presicce - University of Rome, "La Sapienza" - Department of Computer Science
G. Taentzer, Technical University of Berlin, Department of Computer Science
Overview

- Motivations
- Modelling code and specifications
- Formal tools for modeling coordinated transformations
Motivations

• Refactoring becomes more and more popular, one reason: *extreme programming* [Fowler]
• Maintaining consistency of code and specification during refactoring
• Functional equivalence guaranteed, but what about operational equivalence? [Opdyke, Robert]
• Existing approaches consider individual diagrams (e.g. class diagrams, state diagrams).
• How to coordinate the transformations?
• Avoid reverse engineering
A complex refactoring

class Client1 {
    protected Server1 serv1;
    protected Server1 getServer1() { ... }
    protected void exploitService() {
        serv1 = getServer1();
        Object result = serv1.service1(this);
        // code using result
    }
}

class Server1 {
    public void service1(Object requester)
    { ... }
}

class Client2 {
    protected Server2 serv2;
    protected Server2 getServer2() { ... }
    protected void exploitService() {
        serv2 = getServer2();
        Object result = serv2.service2(this);
        // code using result
    }
}

class Server2 {
    public void service2(Object requester)
    { ... }
}
Abstracting clients

abstract class AbstractClient {
    Server serv;
    abstract Server getServer();
    void exploitService() {
        serv = getServer();
        Object result = serv.service(this);
        exploit(result);
    }
    abstract void exploit(Object arg);
}

class Client1 extends AbstractClient {
    Server getServer() {
        // code from getServer1();
    }
    void exploit(Object arg) {
        // previous code using result
    }
}

class Client2 extends AbstractClient {
    Server getServer() {
        // code from getServer2();
    }
    void exploit(Object arg) {
        // previous code using result
    }
}
### Abstracting servers

```java
interface Server {
    void service(Object requester);
}

class Server1 implements Server {
    void service(Object requester) {
        service1(requester);
    }
    void service1(Object requester) {
        ...  
    }
}

class SpecServer2 implements Server {
    void service(Object requester) {
        service2(requester);
    }
    void service2(Object requester) {
        ...  
    }
}
```
Refactorings involved

- extract_code_as_method
  (extract code to `exploitService(object arg)``
- move_method
  (`exploitService()` moved to super class)
- rename_method
  (`getServer1()` and `getServer2()` renamed to `getServer()`)
- rename_inst_variable and pull_up_inst_variable
  (serv1 and serv2 moved to AbstractClient as serv)
Diagram transformations
(extract_code_as_method)

Class diagram:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client1</strong></td>
<td></td>
</tr>
<tr>
<td>serv : SpecServer1</td>
<td></td>
</tr>
<tr>
<td>exploitService() : void</td>
<td></td>
</tr>
<tr>
<td>getServer1() : SpecServer1</td>
<td></td>
</tr>
<tr>
<td>exploit() : Object</td>
<td></td>
</tr>
</tbody>
</table>

Sequence diagram:

:Client1

serv : SpecServer1

exploitService():

serv = getServer1()
result = service1()
exploit(result)
## Refactorings

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Affected Diagram</th>
<th>Creation/Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Class</td>
<td>Class</td>
<td>State</td>
</tr>
<tr>
<td>Remove Class</td>
<td>Class, Sequence</td>
<td>State</td>
</tr>
<tr>
<td>Rename Class</td>
<td>Class, Sequence</td>
<td>-</td>
</tr>
<tr>
<td>Remove Method</td>
<td>Class, Sequence, State</td>
<td>-</td>
</tr>
<tr>
<td>Add Parameter to Method</td>
<td>Class</td>
<td>-</td>
</tr>
<tr>
<td>Extract Code As Method</td>
<td>Class, Sequence, State</td>
<td>-</td>
</tr>
<tr>
<td>Push Up/Down Method</td>
<td>Class, Sequence, State</td>
<td>-</td>
</tr>
</tbody>
</table>
Need for coordination

• Code and diagrams are different views of a software system
• Modifications of the system are to be reflected in the views in a consistent way
• Characteristics of Refactoring:
  – changes in code affect several views
  – no diagram changes without code changes
Idea

• Code and diagrams are abstracted to graphs
• Overlapping in common elements expressed by interface graphs
• Coordinated refactoring by distributed graph transformation
• Complex refactorings by transformation units
Distributed typed graph

- Distributed graph
- Typed graphs

T(F)
Type graphs

For code representation:

For diagrams: UML meta model
Type graph for network graphs
Distributed graph transformation

- DPO Approach
- Rewriting occurs at two levels
Code transformation
(extract_code_as_method)

Attribute condition: \( x, y \geq \textit{first} \) or \( x, y \leq \textit{last} \)
Class diagram transformation
(extract_code_as_method)

1: Class
name = \textit{cname}

2: Method
name = \textit{morig}

: Method
name = \textit{mnew}

: Method
name = \textit{mnew}

: Class

\textit{owner}

\textit{gen}

1: Class
name = \textit{cname}

2: Method
name = \textit{morig}

: Method
name = \textit{mnew}

: Method
name = \textit{mnew}

: Class

\textit{owner}

\textit{gen}
Transitive Closure of Generalization

1: Class
name = \textit{cname}

2: Generalization

3: Class

insert\_gen(string \textit{cname})

1: Class
name = \textit{cname}

2: Generalization
generalization

3: Class
generalization

4: Class
generalization

compute\_gen()

1: Class
generalization

2: Class
generalization

3: Generalization

4: Class
generalization
Sequence diagram transformation

1: Message
2: CallAction
3: Operation
4: Method
name=morig
5: ClassifierRole
receiver
6: Class
name = cname

pre

1: Message
2: CallAction
3: Operation
4: Method
name=morig
5: ClassifierRole
receiver
6: Class
name = cname

succ

sender

base
Statechart diagram transformation

1: State
   Name = sname

2: Transition
   trigger
   target

3: CallEvent
   referred

4: Operation
   name = morig
   trigger
   referred
   owner

5: Class
   name = cname

1: State
   Name = sname

2: Transition
   target
   source

3: CallEvent
   referred

4: Operation
   name = morig
   trigger
   source
   target

5: Class
   name = cname

: Event
   name = „completed“

: Operation
   name = mnew
   owner

: State
   trigger
   source
   target

: CallEvent
   referred
   owner
Transformation units for distributed graph transformations

- Rule expressions to define sequences of applications
- Parameters can be passed into rules
- Expression asLongAsPossible
- Rules applied on each diagram of a given type
- Rules applied on each occurrence of an antecedent in a diagram
- Centralised control on distributed graphs
Sample transformation unit: ExtractCodeAsMethod

**Approach:** DPO on distributed typed graphs

**Initial and terminal graph classes:** all distributed graphs over the given distributed type graph

```java
ExtractCodeAsMethod(String cname, String morig, String mnew, int first, int last):
    asLongAsPossible insert_gen() end;
    asLongAsPossible compute_gen() end;
    asLongAsPossible ecamAll(cname, morig, mnew, first, last) end;
    asLongAsPossible completeEcamSequence(cname, morig, mnew) end;
    asLongAsPossible remove_gen() end
```
Related work

• [Mens99] (type graph for code)
• [MDJ02] (modeling of program transformations)
• [GMT99] (distributed graphs for coordinating views)
• [NWZ01] (reverse engineering with Fujaba)
Conclusions

• Description of software refactoring by distributed graph transformation
• Different kinds of control:
  • negative application conditions
  • coordinating transformations over different diagrams
  • specialisations of rule schemes
  • iteration of rule application
• User chooses refactoring, thus triggering a transformation unit
• No implementation currently available