A Graph Rewriting Formalism for Object-Oriented Software Refactoring

Tom Mens (tom.mens@vub.ac.be)  
Postdoctoral Fellow – Fund for Scientific Research (Flanders)  
Programming Technology Lab  
Vrije Universiteit Brussel
Refactorings are software transformations that restructure an object-oriented application while preserving its behaviour.

According to Fowler (1999), refactoring:
- improves the design of software
- makes software easier to understand
- helps you find bugs
- helps you program faster
Goal

- Improve tool support for refactoring object-oriented software …
  - more scalable (e.g., composite refactorings)
  - more language independent
  - provably correct (e.g., guarantee behaviour preservation)
- … by providing a formal model in terms of
  - graphs
    - compact and expressive representation of program structure and behaviour
    - 2-D nature removes redundancy in source code (e.g., localised naming)
  - graph rewriting
    - intuitive description of transformation of complex graph-like structures
    - theoretical results help in the analysis of such structures
      - (confluence property, parallel/sequential independence, critical pair analysis)
Goal: show feasibility of graph rewriting formalism to express and detect various kinds of behaviour preservation.
UML class diagram

Node

- nextNode
- name
- accept(p:Packet)
- send(p:Packet)

Packet

- originator
- addressee
- contents

Workstation

- originate(p:Packet)

PrintServer

- print(p:Packet)

FileServer

- save(p:Packet)
public class Node {
    public String name;
    public Node nextNode;
    public void accept(Packet p) {
        this.send(p);
    }
    protected void send(Packet p) {
        System.out.println(name + "sends to" + nextNode.name);
        nextNode.accept(p);
    }
}

public class Packet {
    public String contents;
    public Node originator;
    public Node addressee;
}

public class Printserver extends Node {
    public void print(Packet p) {
        System.out.println(p.contents);
    }
    public void accept(Packet p) {
        if(p.addressee == this)
            this.print(p);
        else
            super.accept(p);
    }
}

public class Workstation extends Node {
    public void originate(Packet p) {
        p.originator = this;
        this.send(p);
    }
    public void accept(Packet p) {
        if(p.originator == this)
            System.err.println("no destination");
        else super.accept(p);
    }
}
Two selected refactorings

- **Encapsulate Field**
  - encapsulate public variables by making them private and providing accessor methods
  - Examples
    - `EncapsulateField(name, String getName(), setName(String))`
    - `EncapsulateField(nextNode, Node getNextNode(), setNextNode(Node))`
  - Preconditions
    - accessor method signatures should not exist in inheritance chain

- **Pull up method**
  - move similar methods in subclasses to common superclass
  - Preconditions
    - method to be pulled up should not refer to variables defined in subclass, and its signature should not exist in superclass
public class Node {
    private String name;
    private Node nextNode;
    public String getName() {
        return this.name;
    }
    public void setName(String s) {
        this.name = s;
    }
    public Node getNextNode() {
        return this.nextNode;
    }
    public void setNextNode(Node n) {
        this.nextNode = n;
    }
    public void accept(Packet p) {
        this.send(p);
    }
    protected void send(Packet p) {
        System.out.println(
            name +
            "sends to" +
            nextNode.name);
        nextNode.accept(p);
    }
}
Behaviour preservation

- Only look at static structure of a program

- Many different kinds of preservation
  - Access preserving
    - each method body (transitively) accesses at least the same variables as it did before the refactoring
  - Update preserving
    - each method body (transitively) performs at least the same variable updates as it did before the refactoring
  - Call preserving
    - each method body (transitively) performs at least the same method calls as it did before the refactoring
Graph notation – structure

➢ program structure
Graph notation – behaviour

- behaviour of class Node
## Node type set

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Class</td>
<td>Node, Workstation, PrintServer, Packet</td>
</tr>
<tr>
<td>B</td>
<td>method Body</td>
<td>System.out.println(p.contents)</td>
</tr>
<tr>
<td>V</td>
<td>Variable</td>
<td>name, nextNode, contents, originator</td>
</tr>
<tr>
<td>S</td>
<td>method Signature in lookup table</td>
<td>accept, send, print</td>
</tr>
<tr>
<td>P</td>
<td>formal Parameter of a message</td>
<td>p</td>
</tr>
<tr>
<td>E</td>
<td>(sub)Expression in method body</td>
<td>p.contents</td>
</tr>
</tbody>
</table>
# Edge type set

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l: S \rightarrow B$</td>
<td>dynamic method lookup</td>
<td>class PrintServer \texttt{extends} Node</td>
</tr>
<tr>
<td>$i: C \rightarrow C$</td>
<td>inheritance</td>
<td></td>
</tr>
<tr>
<td>$m: V</td>
<td>B \rightarrow C$</td>
<td>class membership</td>
</tr>
<tr>
<td>$t: P</td>
<td>V</td>
<td>S \rightarrow C$</td>
</tr>
<tr>
<td>$p: S \rightarrow P$</td>
<td>formal parameter</td>
<td>send(Packet \texttt{p})</td>
</tr>
<tr>
<td>$p: E \rightarrow E$</td>
<td>actual parameter</td>
<td>System.out.println(nextNode.name)</td>
</tr>
<tr>
<td>$e: B \rightarrow E$</td>
<td>expression in method body</td>
<td></td>
</tr>
<tr>
<td>$\cdot: E \rightarrow E$</td>
<td>cascaded expression</td>
<td>nextNode.accept(p)</td>
</tr>
<tr>
<td>$d: E \rightarrow S$</td>
<td>dynamic method call</td>
<td>this.send(p)</td>
</tr>
<tr>
<td>$a : E \rightarrow P</td>
<td>V$</td>
<td>access of parameter of variable</td>
</tr>
</tbody>
</table>
Well-formedness constraints

- Use type graph
Well-formedness constraints

- Use *forbidden subgraphs*
  - WF-1: a variable with the same name cannot be defined twice in the same inheritance hierarchy
  - WF-2: a method with the same signature cannot be implemented twice in the same class
  - WF-3: a method cannot refer to variables in descendant classes

```
WF-1

WF-2

WF-3
```
Graph production **EncapsulateField**

- **EncapsulateField**(var, accessor, updater)
  - parameterised production
  - embedding mechanism takes context into account

<table>
<thead>
<tr>
<th>incoming edges</th>
<th>outgoing edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u,1) → (d,2)</td>
<td>(m,1) → (m,1), (m,4), (m,5)</td>
</tr>
<tr>
<td>(a,1) → (d,3)</td>
<td>(t,1) → (t,1), (t,3), (t,6)</td>
</tr>
</tbody>
</table>

![Diagram of EncapsulateField](image-url)
Graph production *EncapsulateField*

- Application of the production in the context of the LAN simulation
  - *EncapsulateField*(name, getName, setName)
Graph production \textit{PullUpMethod}

- \textit{PullUpMethod}(parent, child, name)
  - has an effect on all subclasses
  - \textit{controlled graph rewriting} needed

\begin{itemize}
  \item \textbf{P}_1 \quad \text{parent } C \quad \text{child } C \quad \text{B} \quad \text{name } S
  \begin{array}{c}
    1 \quad \text{parent } C \quad \text{child } C \quad \text{B} \quad \text{name } S
  \end{array}
  \begin{array}{c}
    \uparrow \text{i} \quad \uparrow \text{m} \quad \downarrow \text{l}
  \end{array}
  \rightarrow
  \begin{array}{c}
    \quad \text{parent } C \quad \text{child } C \quad \text{B} \quad \text{name } S
  \end{array}
  \begin{array}{c}
    \quad \uparrow \text{i} \quad \quad \downarrow \text{m} \quad \quad \downarrow \text{l}
  \end{array}

  \item \textbf{P}_2 \quad \text{parent } C \quad \text{child } C \quad \text{B} \quad \text{name } S
  \begin{array}{c}
    \quad \text{parent } C \quad \text{child } C \quad \text{B} \quad \text{name } S
  \end{array}
  \begin{array}{c}
    \quad \uparrow \text{i} \quad \quad \downarrow \text{m} \quad \quad \downarrow \text{l}
  \end{array}
\end{itemize}
Access preserving

- Use graph expression

- EncapsulateField preserves behaviour
  - access preserving: all attribute nodes can still be accessed via a transitive closure.
Update preserving

- Use graph expression

- EncapsulateField preserves behaviour
  - update preserving: all attribute nodes can still be updated via a transitive closure.
Call preserving

- Use graph expression

PullUpMethod preserves behaviour

Call preserving:
Satisfying preconditions

- All refactorings must satisfy well-formedness conditions
  - WF-1, WF-2, WF-3
- Some refactorings require additional constraints
  - E.g. *EncapsulateField* may not introduce accessor/updater method if their signatures are defined in the inheritance chain (RC1)
- Use *negative application preconditions* to fulfill these constraints
  - E.g., for *EncapsulateField*
Conclusion

- graph rewriting suitable for specifying effect of refactorings
  - language-independent
  - natural and precise way to specify transformations
  - behaviour preservation can be formally verified
- better integration of existing graph techniques needed
  - well-formedness constraints
    - type graphs, forbidden subgraphs
  - infinite sets of productions
    - parameterisation and embedding mechanism
  - restricting applicability
    - negative preconditions and controlled graph rewriting
Future work

- more validation
  - more refactorings
  - more case studies
  - more kinds of behaviour preservation
- further work on formalism
  - apply approach to arbitrary evolution steps
  - investigate language independence and scalability
  - difficult to manipulate nested structures
    - e.g. copying or moving entire method body
- tool support
- classification of refactorings based on
  - preservation properties
  - complexity of the refactoring