A Graph Rewriting Model for Object-Oriented Software Refactoring

Tom Mens, Serge Demeyer, Dirk Janssens

{ tom.mens@vub.ac.be | serge.demeyer | dirk.janssens }@ua.ac.be

Programming Technology Lab
Vrije Universiteit Brussel

Lab on Re-Engineering
Universiteit Antwerpen
What is refactoring?

- 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 …
  - less *ad hoc*
  - more scalable (e.g., composite refactorings)
  - more language independent
  - more 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)
Case study: LAN simulation

- Goal: show feasibility of graph rewriting formalism to express and detect various kinds of behaviour preservation.
UML class diagram

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

Packet
- contents
- addressee
- originator

Workstation
- accept(p:Packet)
- originate(p:Packet)

PrintServer
- accept(p:Packet)
- print(p:Packet)

FileServer
- accept(p:Packet)
- 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);
    }
}
Graph representation – part 1

➢ program structure

- println $M$  (s) $P$
- send $M$  (p) $P$
- accept $M$  (p) $P$
- originate $M$  (p) $P$
- print $M$  (p) $P$

- String $C$
- contents $A$
- originator $A$
- addressee $A$

- Node $C$

- Workstation $C$
- (origin) $B$
- (accept) $B$

- PrintServer $C$
- (accept) $B$
- (print) $B$

- Packet $C$

FWO-WOG, January 2002, Brussels © Tom Mens, Vrije Universiteit Brussel
Graph representation – part 2

- program behaviour for class *Workstation*

```java
void originate(Packet p)
1:{ p.originator = this;
2: this.send(p); }
```
Refactoring – Encapsulate Field

Fowler 1999, page 206

*There is a public field*

Make it private and provide accessors

```java
class Node {
    private String name;
    private Node nextNode;
    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);
    }
}
```

```java
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(
            this.getName() +
            "sends to" +
            this.getNextNode().getName());
        this.getNextNode().accept(p);
    }
}
```
Refactoring – Encapsulate Field

➢ before the refactoring
Refactoring – Encapsulate Field

➤ after the refactoring
refactoring is achieved by applying two occurrences of production \textit{EncapsulateField}(\textit{class,attr,type,accessor,updater})

- EncapsulateField(Node, name, String, getName, setName)
- EncapsulateField(Node, nextNode, Node, getNextNode, setNextNode)
Behaviour preservation invariants

- Access preservation
  - each method body (indirectly) performs at least the same attribute accesses as it did before the refactoring

- Update preservation
  - each method body (indirectly) performs at least the same attribute updates as it did before the refactoring

- Statement preservation
  - each method (indirectly) performs at least the same statements as it did before the refactoring

- Type preservation
  - each statement in each method body still has the same result type or return type as it did before the refactoring
Behaviour preservation invariants

- **EncapsulateField preserves behaviour**
  - *access preserving*: all attribute nodes can still be accessed via a transitive closure
    - 
  - *update preserving*: all attribute nodes can still be updated via a transitive closure
    - 

Behaviour preservation invariants can be detected by *graph patterns*
Conclusion

- Graph rewriting seems a useful and promising formalism to provide support for refactoring
  - More practical validation needed
  - Current experiment only focuses on behaviour preservation
  - A formalism can assist the refactoring process in many other ways

- Proposed FWO research project (4 years / 3 persons)
Open questions

- Which program properties should be preserved by refactorings?
  - input/output behaviour, timing constraints, static versus dynamic behaviour
- What is the complexity of a refactoring?
  - complexity of applicability / complexity of applying the refactoring
- How do refactorings affect quality factors?
  - increase/decrease complexity, understandability, maintainability, …
- How can refactorings be composed/decomposed?
  - composite refactorings / extracting refactorings from successive releases
- How do refactorings interact?
  - parallel application of refactorings may lead to consistency problems
- How to provide support for non-behaviour-preserving refactorings?
- Co-evolution: How do refactorings affect design models?
- Language-independent formalism for refactoring?