Using Graph Rewriting Models for Object-Oriented Software Evolution

Tom Mens

tom.mens@vub.ac.be
Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Object-oriented software evolution

- Need better tool support for
  - version control
    - e.g. upgrading application frameworks
  - collaborative software development
    - e.g. software merging
  - evolution at a higher level of abstraction
    - e.g. refactoring
    - e.g. evolution of design pattern (instances)
  - change propagation, change impact analysis, effort estimation
  - ...

Seminaire CERTI, 8/3/2002 UCL © Tom Mens, Vrije Universiteit Brussel
Tool support must be

- scalable
  - applicable to large-scale industrial software systems
  - “A major challenge for the research community is to develop a good theoretical understanding an underpinning for maintenance and evolution, which scales to industrial applications.” [Bennett & Rajlich 2000]

- generic
  - independent of the programming or modelling language
  - applicable in all phases of the software life-cycle

- formally founded
  - to prove that results are well-formed, correct, complete, confluent, compositional, ...
Graph rewriting

- can be used as formal model for software evolution

- 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, compositionally, …
Two applications

- Software refactoring
  - use graph rewriting to express and detect various kinds of behaviour preservation
    - in collaboration with Prof. Serge Demeyer and Prof. Dirk Janssens, University of Antwerp

- Software merging
  - use graph rewriting to detect evolution/merge conflicts between parallel evolutions of the same software
    - PhD thesis of Tom Mens
Application 1

Software Refactoring
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

- Formalisms can help to
  - gain insight in the fundamental underlying principles
  - prove correctness, e.g., to guarantee behaviour preservation
Refactoring case study: LAN

- Goal: show feasibility of graph rewriting formalism to express and detect various kinds of behaviour preservation
```java
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);
    }
}
```
Refactoring Example 1: Encapsulate Field

Fowler 1999, page 206

There is a public field

Make it private and provide accessors

```java
public class Node {
    private String name;
    private 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 Example 2: Extract Method

Fowler 1999, page 110

You have a code fragment that can be grouped together

Turn the fragment into a method whose name explains the purpose of the method

```java
public class Node {

    ... 

    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); }
}
```

```java
public class Node {

    ... 

    public void accept(Packet p) {
        this.send(p); }
    protected void send(Packet p) {
        this.log(p);
        this.getNextNode().accept(p); }
    protected void log(Packet p) {
        System.out.println(
            this.getName() + "sends to" +
            this.getNextNode().getName());
        this.getNextNode().accept(p); }
}
```
Graph representation – part 1

➢ program structure

- Println \( M \)
- Send \( M \)
- Accept \( M \)
- Originate \( M \)
- Print \( M \)
- String \( C \)
- Contents \( A \)
- Originator \( A \)
- Addressee \( A \)
- Node \( C \)
- Workstation \( C \)
- PrintServer \( C \)

Annotations:
- (p) \( P \)
- (s) \( P \)
Graph representation – part 2

➢ program behaviour for class Node

```java
void send(Packet p) {
    1: System.out.println(
       name + "sends to" + nextNode.name);
    2: nextNode.accept(p); }
```
Refactoring Example 1: Encapsulate Field

➢ before the refactoring
Refactoring Example 1: Encapsulate Field

➢ after the refactoring

Seminaire CERTI, 8/3/2002 UCL © Tom Mens, Vrije Universiteit Brussel
Refactoring Example 1: EncapsulateField graph production

- refactoring achieved by applying 2 occurrences of production
  `EncapsulateField(class, attr, type, accessor, updater)`
  - EncapsulateField(Node, name, String, getName, setName)
  - EncapsulateField(Node, nextNode, Node, getNextNode, setNextNode)
Refactoring Example 1: Behaviour preservation

- EncapsulateField preserves behaviour

  - *access preserving*: all attribute nodes can still be accessed via a transitive closure

    ![Accessibility Diagram](diagram1)

  - *update preserving*: all attribute nodes can still be updated via a transitive closure

    ![Update Diagram](diagram2)
Refactoring Example 2: Extract Method

➢ before the refactoring: ExtractMethod(Node, send, log, {1})
Refactoring Example 2:

Extract Method

➢ after the refactoring ExtractMethod(Node,send,log,{1})
Refactoring Example 2: ExtractMethod graph production

- refactoring is achieved by applying an occurrence of production ExtractMethod(\textit{class, old, new, StatList})
- given the body of method \textit{old} in \textit{class}, redirect all statements in \textit{StatList} to the body of a parameterless method \textit{new}

\[ (\text{new}) \quad B \quad \downarrow \quad 2 \quad \text{old} \quad M \quad \downarrow \quad 1 \]
\[ (\text{old}) \quad B \quad \downarrow \quad \text{new} \quad M \quad \downarrow \quad 3 \]
\[ \forall \alpha \in \text{StatList} \]

(method parameters can be introduced afterwards by AddParameter)
Refactoring Example 2: Behaviour preservation

- ExtractMethod preserves behaviour

- *statement preserving*: all expressions (calls, accesses, updates) that were performed before the refactoring, are still performed (via transitive closure) after the refactoring

\[ \forall \alpha \in \text{StatList} \]
Behaviour preservation types

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

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

- **Statement preservation** (see ExtractMethod)
  - each method body (indirectly) performs at least the same statements as ...

- **Call preservation**
  - each method body (indirectly) performs at least the same method calls as ...

- **Type preservation**
  - each statement in each method body still has the same result type or return type as ...
Refactoring: 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)
Refactoring: Open questions

- Which program properties should be preserved by refactorings?
  - input/output behaviour, timing constraints, static versus dynamic behaviour
  - support for non-behaviour-preserving refactorings?

- 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 do refactorings affect design models?

- Language-independent formalism for refactoring?
Application 2

Software

Merging
What is Software Merging?

- **Context: Collaborative Software Development**
  - many software developers working together on the same software
  - need to integrate parallel changes made to the same code

- **Software merging**
  - automated tool support for integrating these parallel changes
  - detect inconsistencies (conflicts) between parallel changes
  - provide support for resolving these inconsistencies

- Merge tools are usually part of a configuration management system or version management system
  - e.g. CVS, RCS, ClearCase, Adele, ...
Problem

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 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);
    }
}

Encapsulate Field

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

Current merge tools cannot merge the two parallel changes automatically due to a merge conflict in the send method
Solution

Sequential application of the parallel graph rewritings yields correct merge result.
Express and document software evolution by means of explicit graph transformations

- Detect whether parallel evolutions can be sequentialised (parallel/sequential independence)
- If not, syntactic conflicts need to be resolved first
  - e.g. renaming same entity twice
- If yes, merging corresponds to sequential application of graph transformations
  - Order of application is irrelevant (confluence property)
  - Potential semantic conflicts need to be detected (based on category-theoretical notion of pushouts/pullbacks)
see

- A formal foundation for object-oriented software evolution

- Conditional graph rewriting as a domain-independent formalism for software evolution

- A state-of-the-art survey on software merging
Open Issue: Structural Conflicts

- **InsertClass**
  - (Shape, Quadrangle, [Square, Rectangle])

- **AddSubclass**
  - (Shape, Parallelogram)
  - (Shape, Triangle)

More difficult to detect in a general way
General Conclusion

- Graph rewriting can be used as a formal model for various aspects of software evolution
  - software refactoring
  - software merging
  - ...
- Formalism allows us to express interesting properties
  - Behaviour preservation of refactorings
  - Compositionality of refactorings
  - Syntactic and semantic merge conflicts
- More theoretical research and practical validation needed
  - Proposed FWO research project (4 years / 3 persons)
  - Apply formalism on evolution of real software systems