A Graph Rewriting Formalism for Object-Oriented Software Refactoring

PROG

Tom Mens (tom.mens@vub.ac.be)
Postdoctoral Fellow – Fund for Scientific Research (Flanders)
Programming Technology Lab
Vrije Universiteit Brussel

What is refactoring?

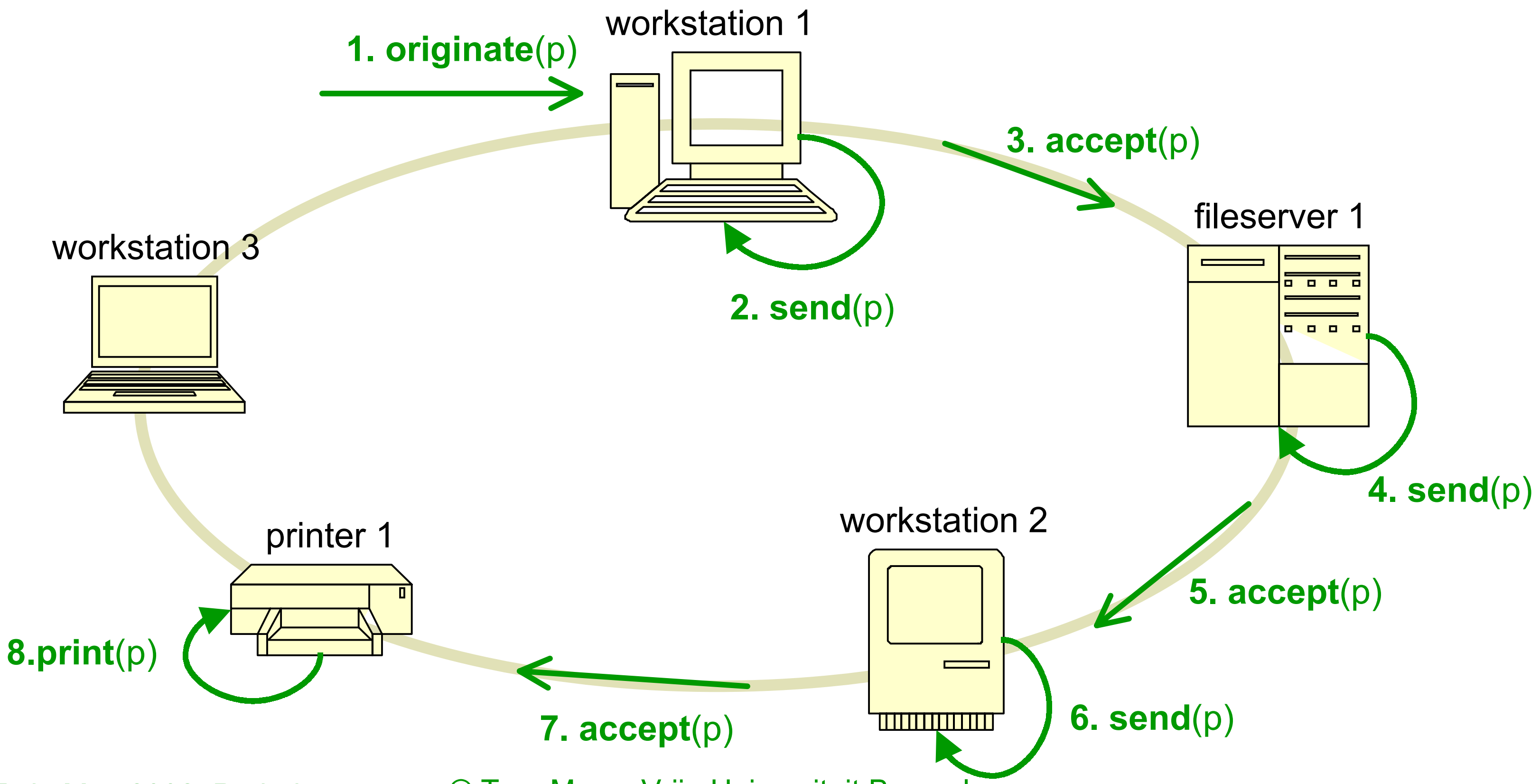
- Refactorings are software transformations that restructure an object-oriented application while preserving its behaviour.
- According to Fowler (1999), refactoring
  - improves the design of software
  - makes software easier to understand
  - helps you find bugs
  - helps you program faster

Goal

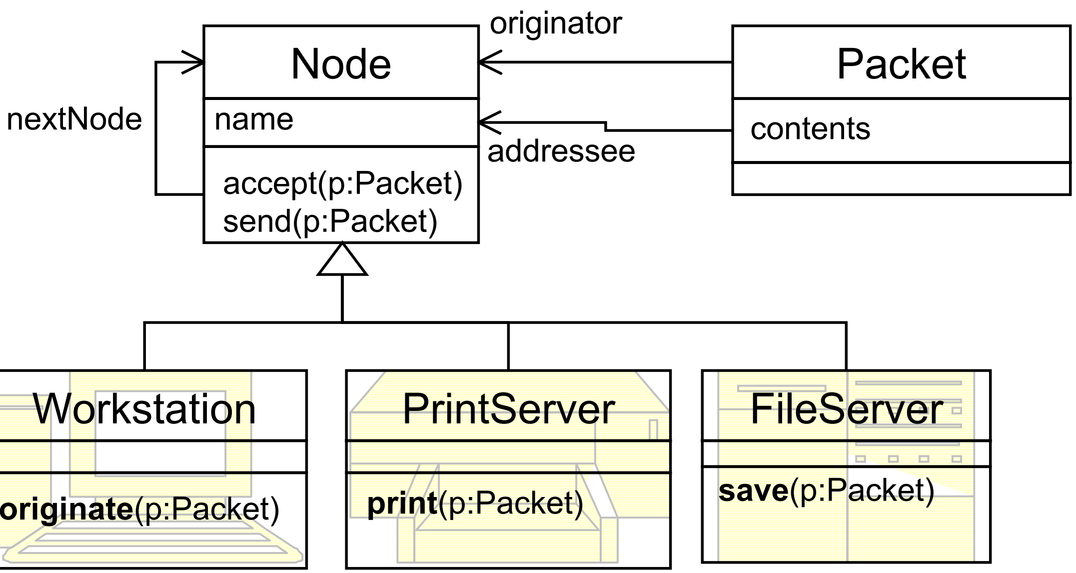
- Improve tool support for refactoring object-oriented software …
  - more scalable (e.g., composite refactorings)
  - more language independent
  - provably correct (e.g., guarantee behaviour preservation)
- … by providing a formal model in terms of
  - graphs
    - compact and expressive representation 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)

Feasibility study: LAN simulation

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



UML class diagram



Java source code

```
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

Refactoring – Encapsulate Field

```
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 {
  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); }
  }
```

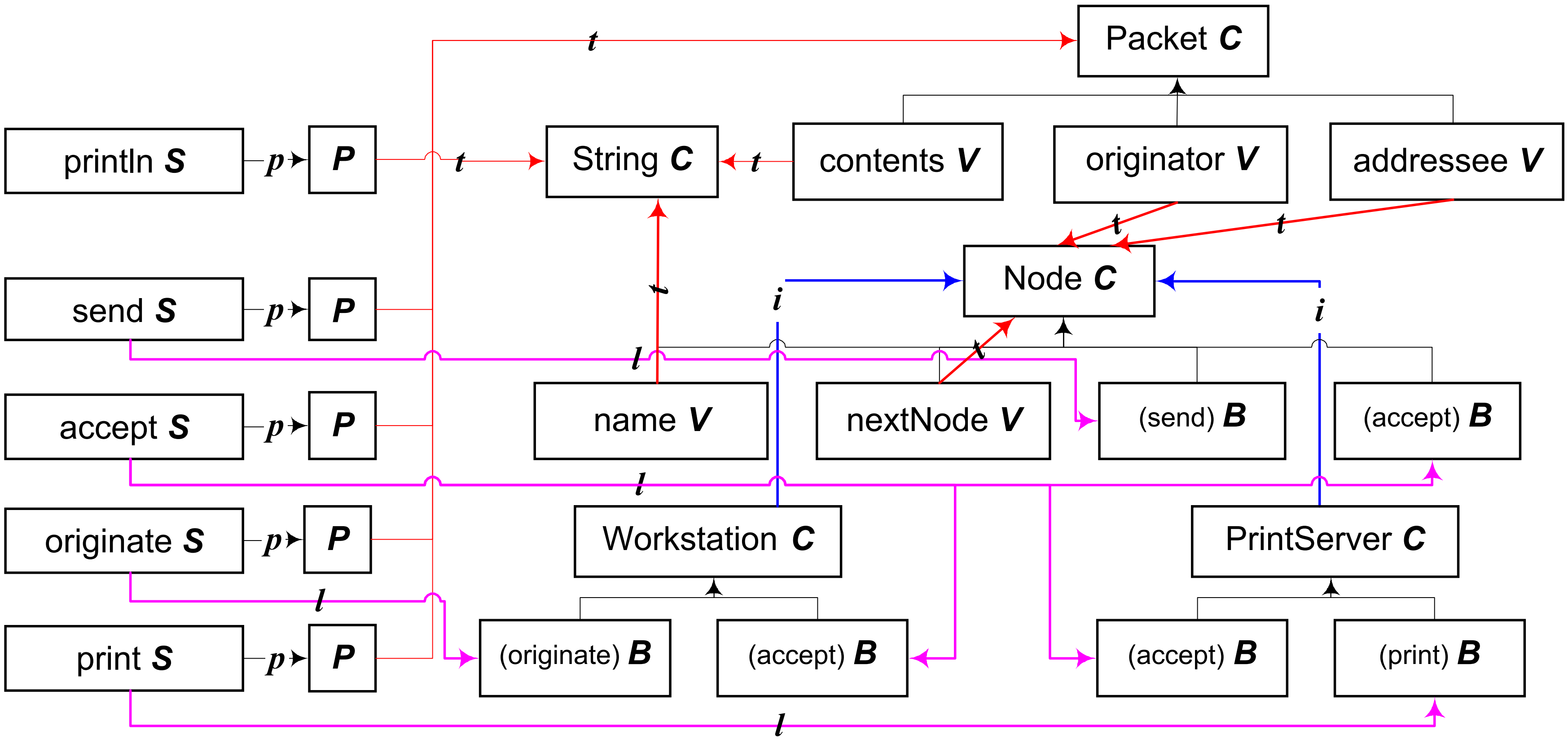
Behaviour preservation

- 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

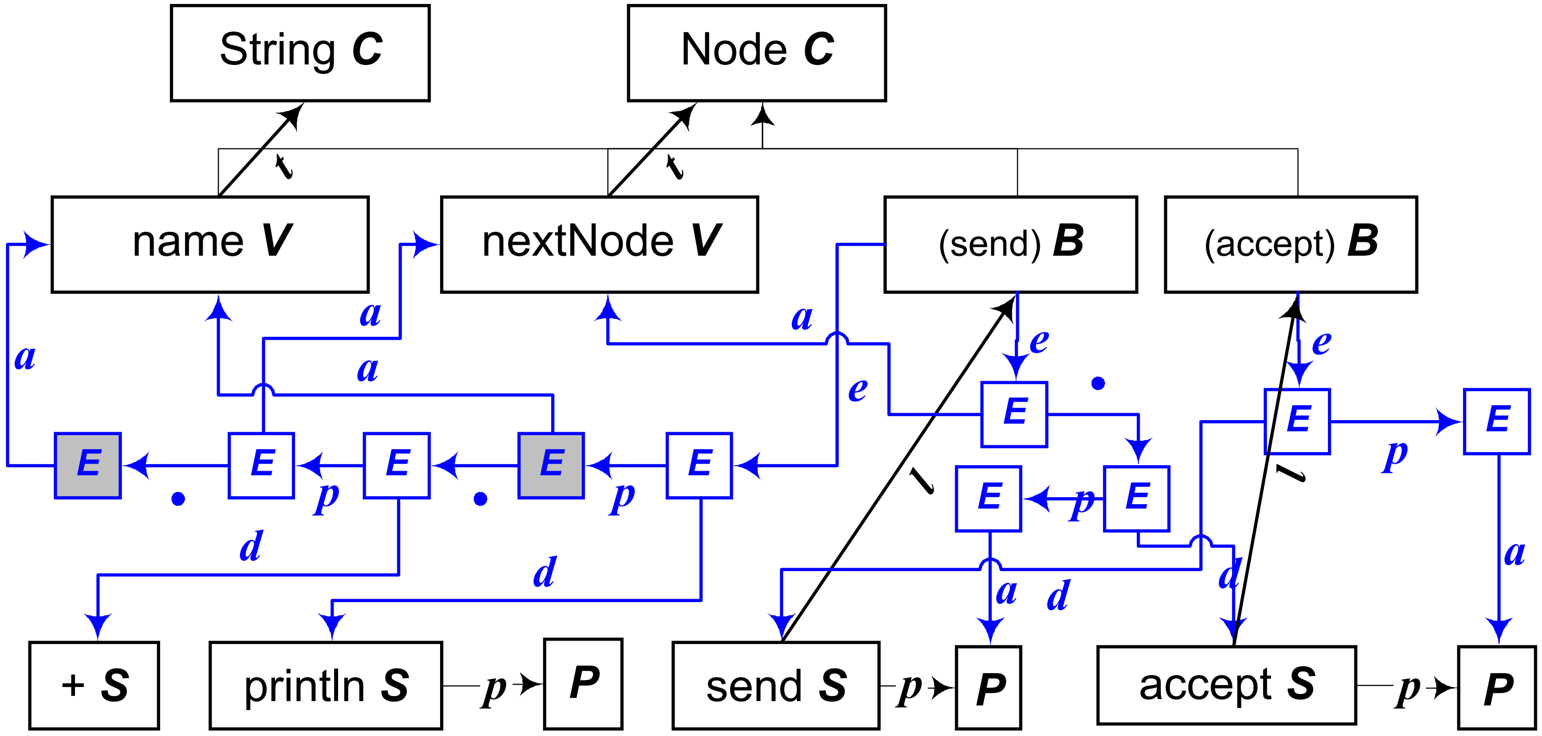
Graph notation – structure

- program structure



Graph notation – behaviour

- behaviour of class Node



Node type set

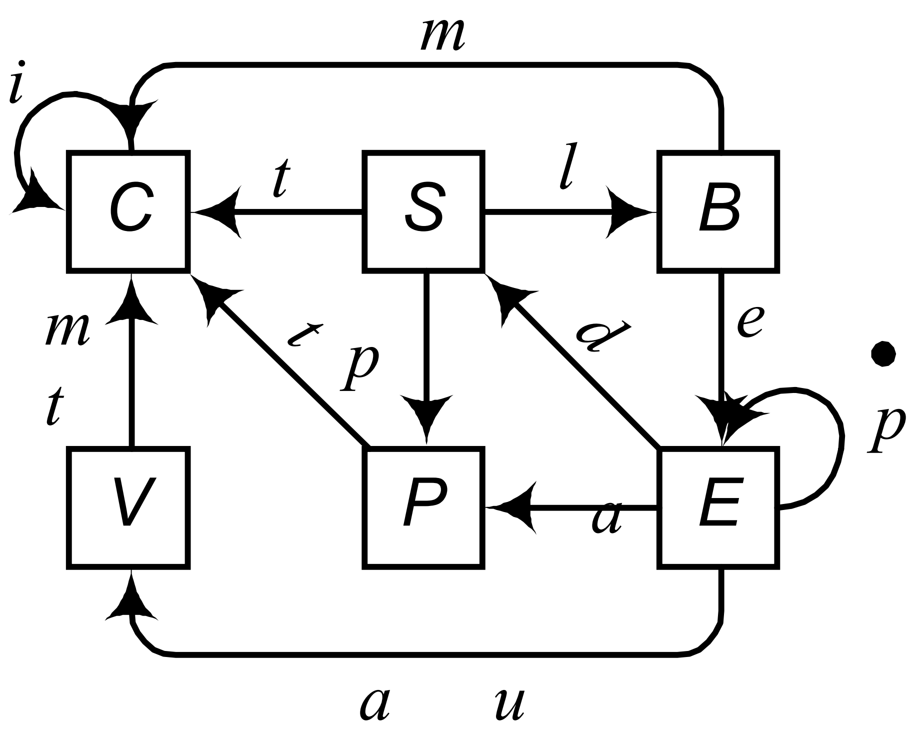
Type	Description	Examples
C	**C**lass	*Node*, *Workstation*, *PrintServer*, *Packet*
B	method **B**ody	`System.out.println(p.contents)`
V	**V**ariable	*name*, *nextNode*, *contents*, *originator*
S	method **S**ignature in lookup table	*accept*, *send*, *print*
P	formal **P**arameter of a message	*p*
E	(sub)**E**xpression in method body	`p.contents`

Edge type set

Type	Description	Examples
$l: S \rightarrow B$	dynamic method **l**ookup	
$i: C \rightarrow C$	**i**nheritance	`class PrintServer` **`extends`** `Node`
$m: V\vert B \rightarrow C$	class **m**embership	
$t: P\vert V\vert S \rightarrow C$	**t**ype	`send(`**`Packet`**` p),` **`String`** `getName()`
$p: S \rightarrow P$	formal **p**arameter	`send(Packet` **`p`**`)`
$p: E \rightarrow E$	actual **p**arameter	`System.out.println(`**`nextNode.name`**`)`
$e: B \rightarrow E$	**e**xpression in method body	
$\bullet: E \rightarrow E$	cascaded expression	`nextNode.accept(p)`
$d: E \rightarrow S$	**d**ynamic method call	`this.send(p)`
$a : E \rightarrow P\vert V$	access of parameter of variable	`p.originator`

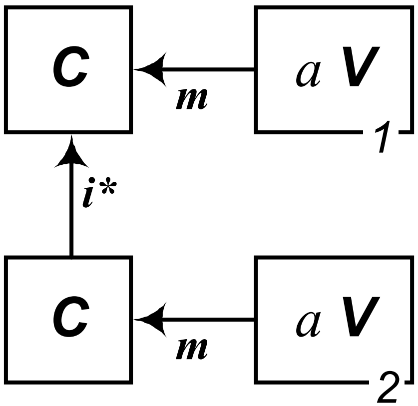
Well-formedness contraints

- Use *type graph*

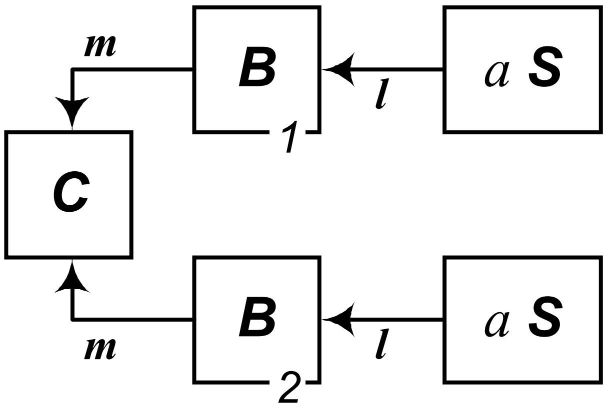


Well-formedness constraints

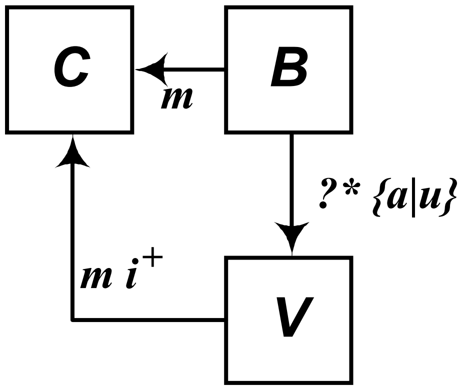
- Use *forbidden subgraphs*
  - WF-1: a variable with the same name cannot be defined twice in the same inheritance hierarchy
  - WF-2: a method with the same signature cannot be implemented twice in the same class
  - WF-3: a method cannot refer to variables in descendant classes



WF-1



WF-2

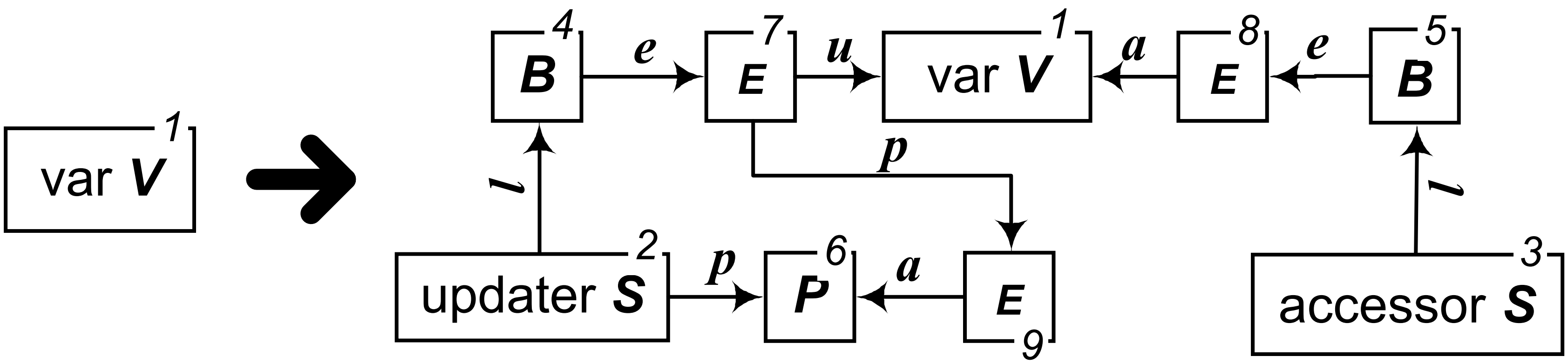


WF-3

Graph production *EncapsulateField*

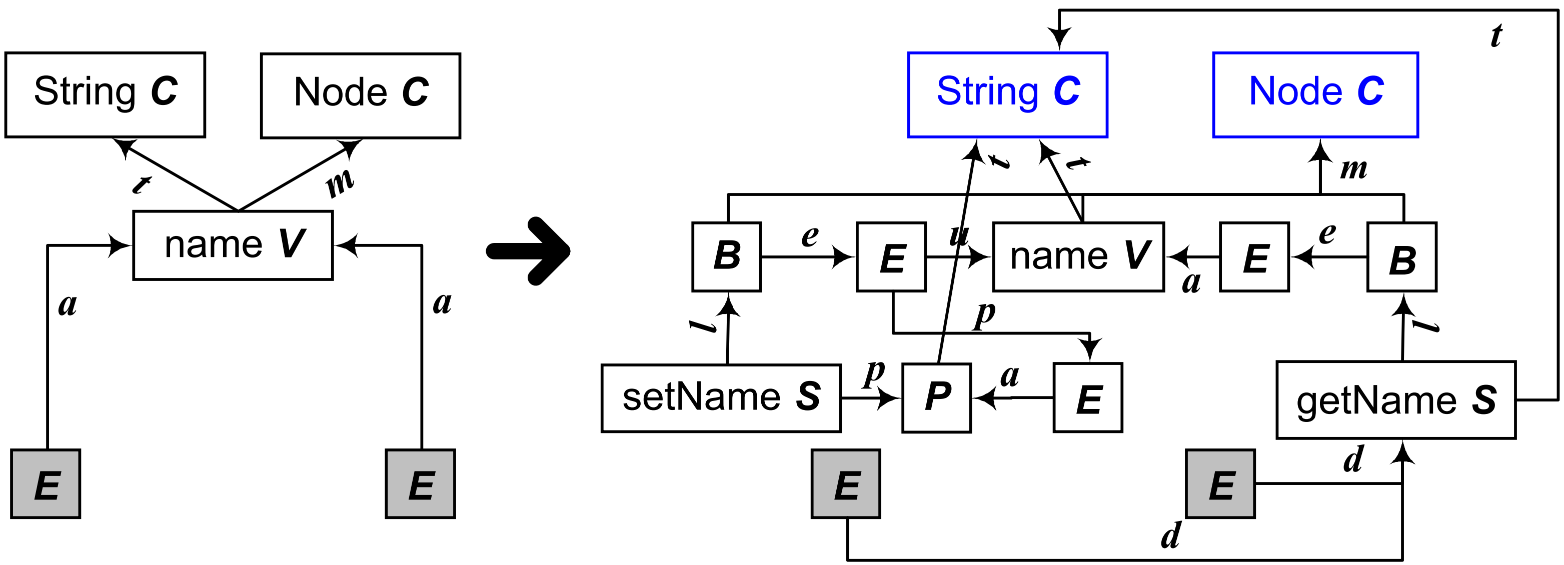
- *EncapsulateField(var,accessor,updater)*
  - *parameterised* production
  - *embedding mechanism* takes context into account

incoming edges	outgoing edges
(u,1) → (d,2)	*(m,1) → (m,1), (m,4), (m,5)*
(a,1) → (d,3)	*(t,1) → (t,1), (t,3), (t,6)*



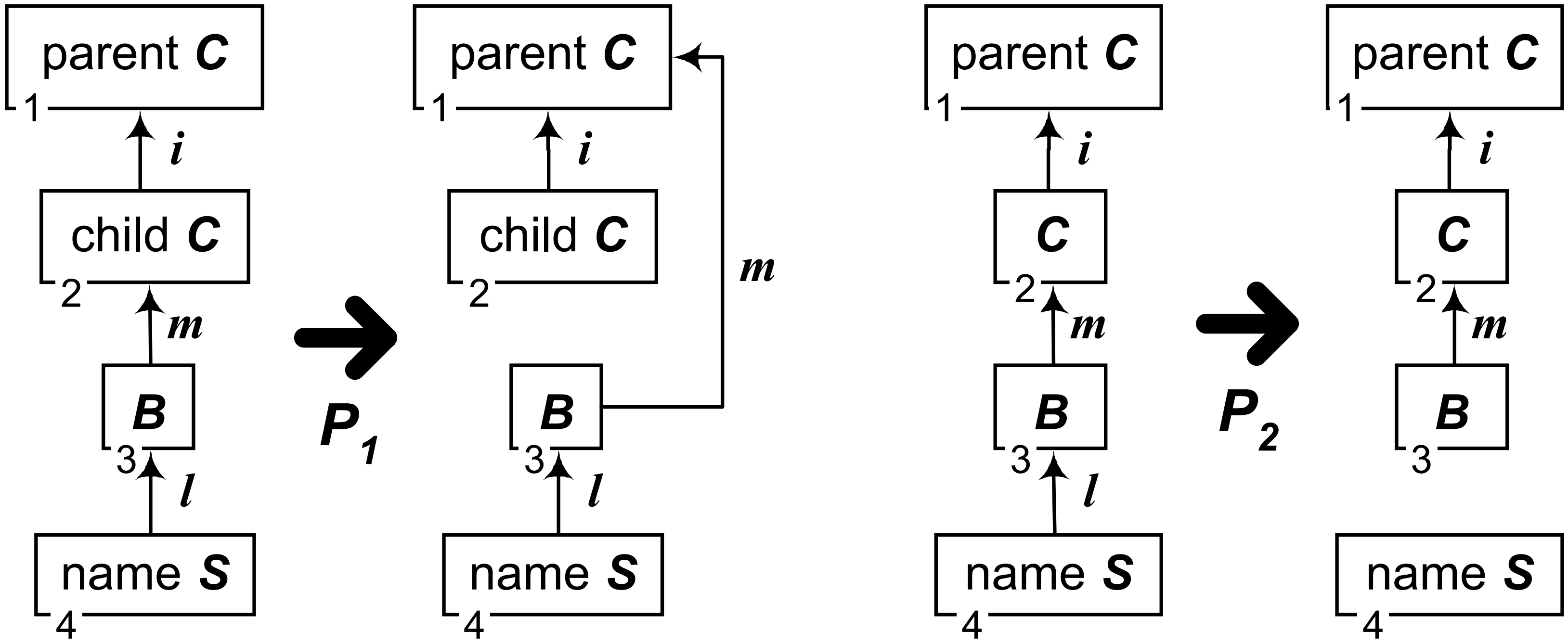
Graph production *EncapsulateField*

- Application of the production in the context of the LAN simulation
  - EncapsulateField(name,getName,setName)



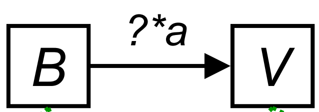
Graph production *PullUpMethod*

- *PullUpMethod(parent,child,name)*
  - has an effect on all subclasses
  - *controlled graph rewriting* needed

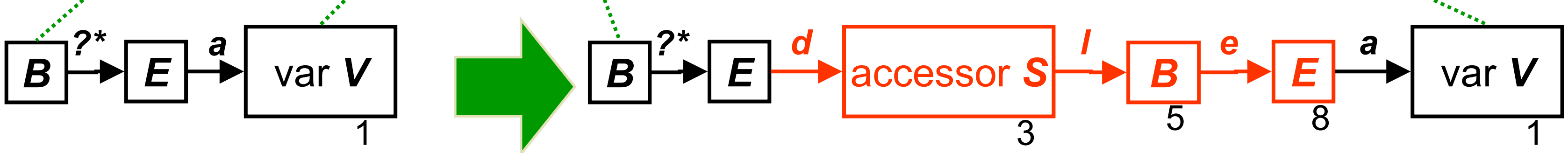


Access preserving

- Use graph expression

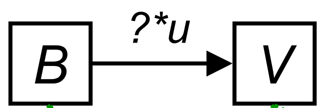


- EncapsulateField preserves behaviour
  - access preserving: all attribute nodes can still be accessed via a transitive closure

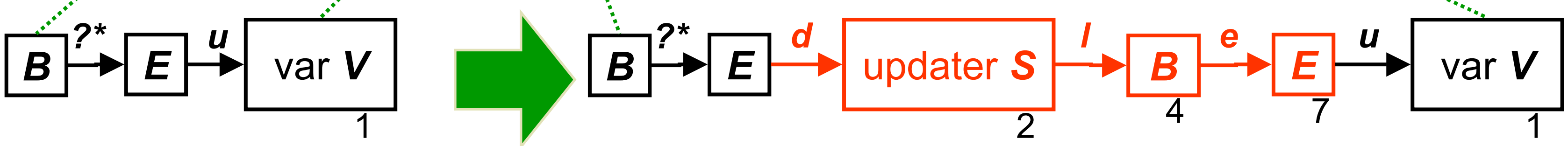


Update preserving

- Use graph expression

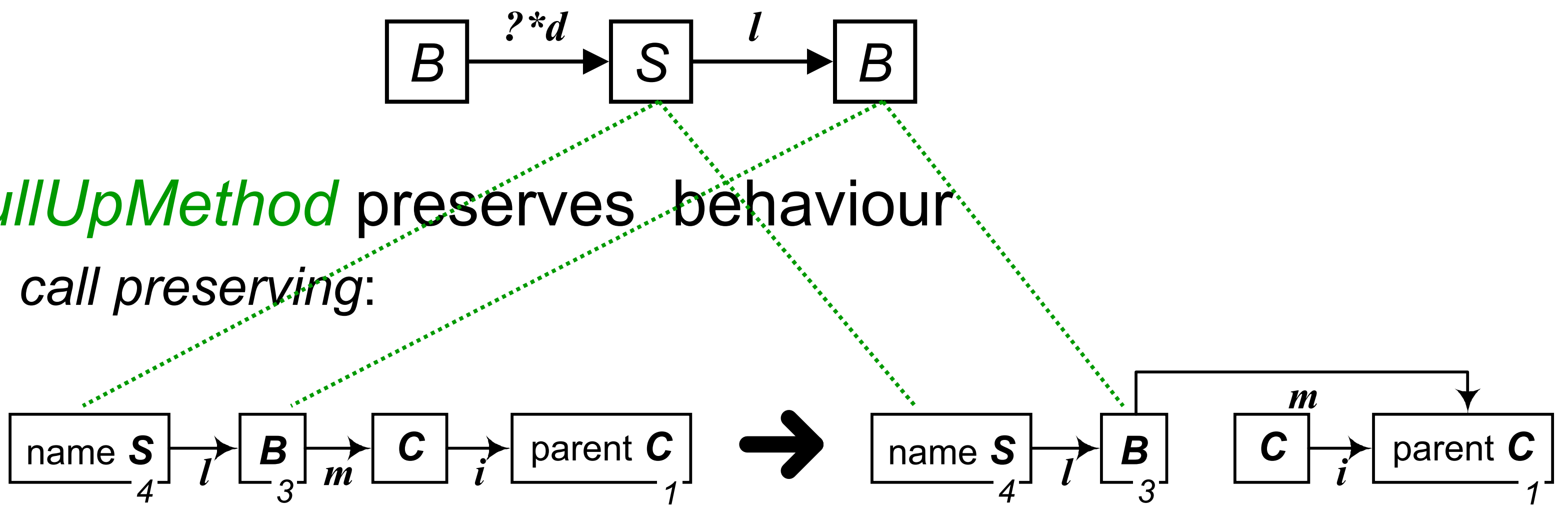


- EncapsulateField preserves behaviour
  - update preserving: all attribute nodes can still be updated via a transitive closure



Call preserving

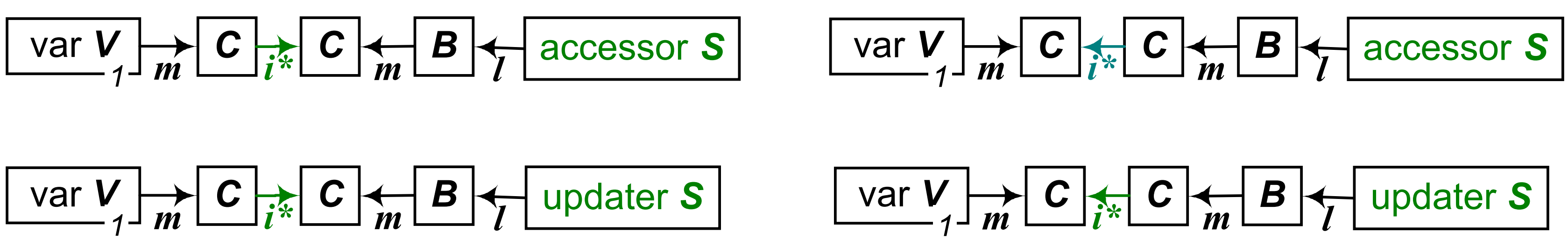
- Use graph expression



- *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