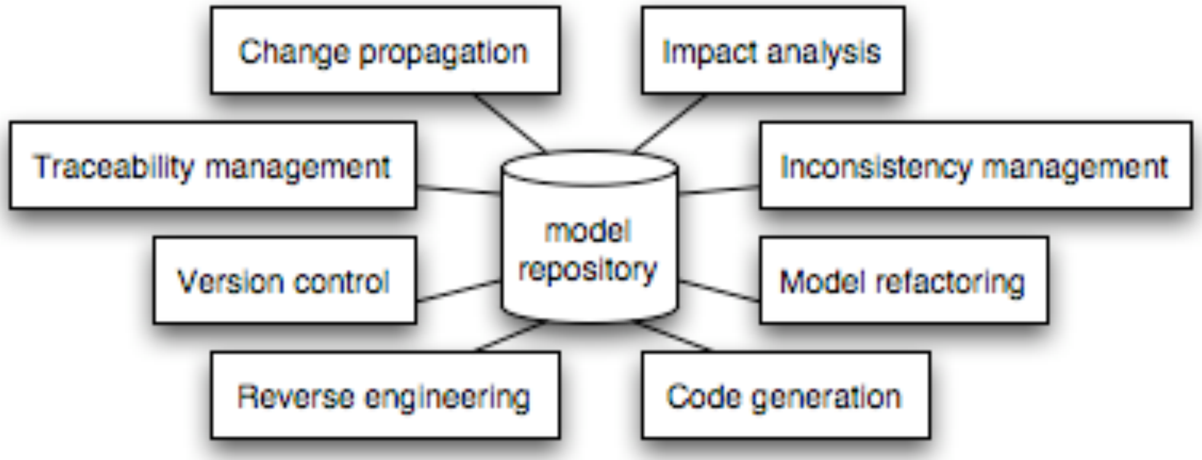
Analysing refactorings with graph transformation theory

Tom Mens

http://w3.umh.ac.be/genlog
Software Engineering Lab
University of Mons-Hainaut
Belgium

Introduction - Software Evolution

- More and better tool support needed for software evolution
  - At all levels of abstraction (e.g. programs and models)
  - For a variety of different activities



Introduction - Software Evolution

- Formalisms can be helpful for such evolution support
  - Description logics
    - For model inconsistency management
      - collaboration with R. Van Der Straeten, VUB
  - Graph transformation
    - For supporting software refactoring
    - Reasoning about preservation properties
      - collaboration with D. Janssens and S. Demeyer, UA
    - Analysing refactoring dependencies
      - collaboration with G. Taentzer and O. Runge, TU Berlin

Graph transformations

- GT theory theoretical results can help during analysis of model refactorings
  - type graph, negative application conditions, parallel and sequential (in)dependence, confluence and critical pair analysis
- GT tools allow us to perform concrete experiments
  - *AGG* (in collaboration with Berlin)

- Current focus
  - *Analysing dependencies between class diagram refactorings*

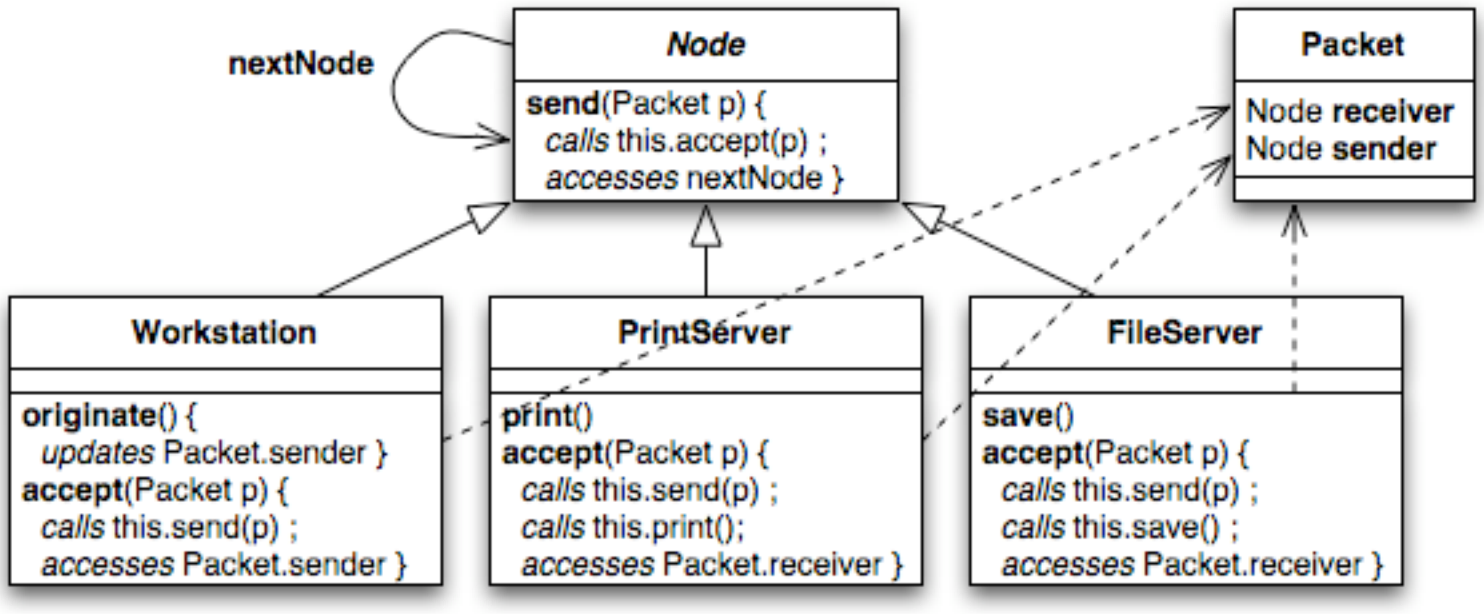
Analysing refactoring dependencies

- Concrete Scenario: Suggest refactoring opportunities
  - What are the alternatives of a selected refactoring?
  - Which other refactorings need to be applied first in order to make the selected refactoring applicable?
  - Which other refactorings are still applicable after applying the selected refactoring?

- Goal: Automate the detection of
  - mutual exclusion relationships between refactorings
  - sequential dependencies between refactorings

Analysing refactoring dependencies

- Example



Analysing refactoring dependencies

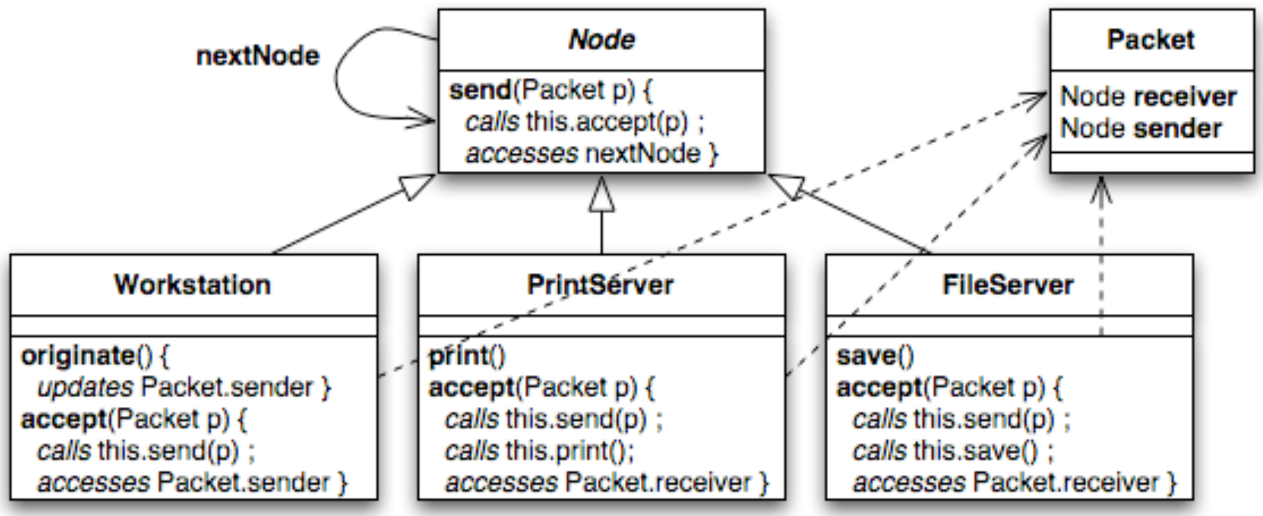
- Refactoring opportunities

  T1 Rename Method print in PrintServer to process

  T2 Rename Method save in FileServer to process

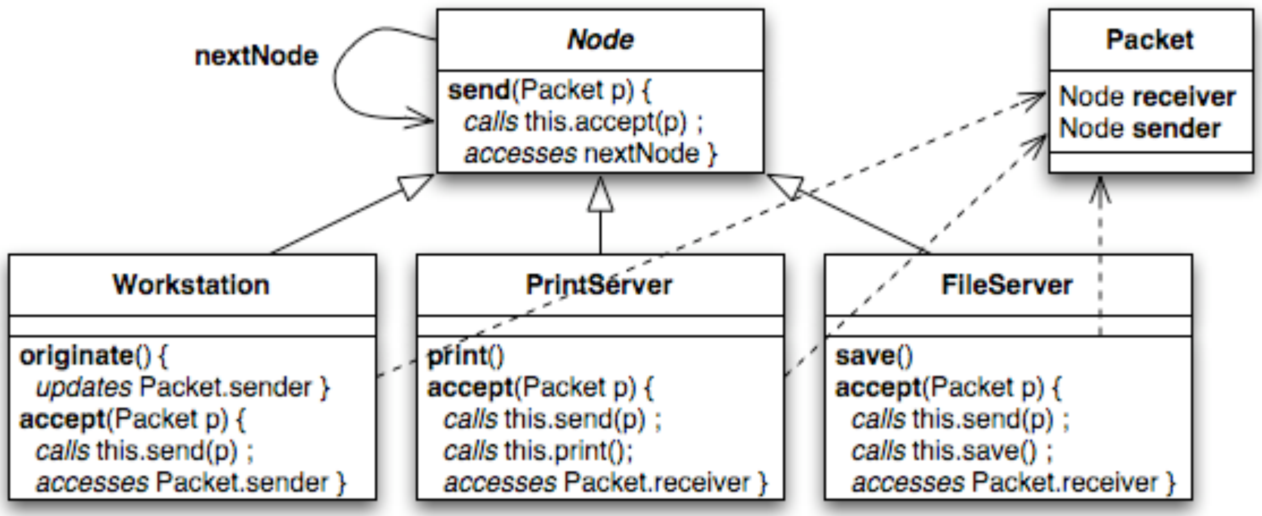
  T3 Create Superclass Server for PrintServer and FileServer

  T4 Pull Up Method accept from PrintServer and FileServer to Server



Analysing refactoring dependencies

- Refactoring opportunities
  - T5 Move Method accept from PrintServer to Packet
  - T6 Move Method accept from FileServer to Packet
  - T7 Encapsulate Variable receiver in Packet
  - T8 Add Parameter p of type Packet to method print in PrintServer
  - T9 Add Parameter p of type Packet to method save in FileServer

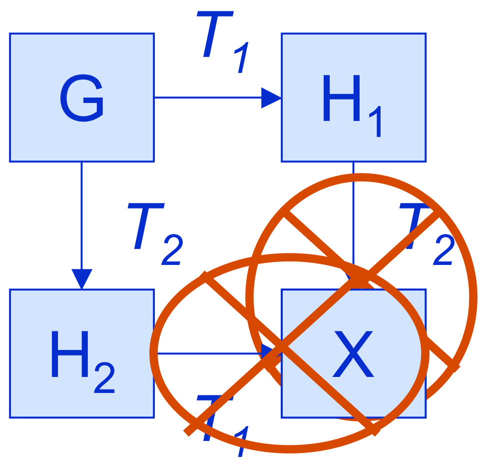


Analysing refactoring dependencies

	T1	T2	T3	T4	T5	T6	T7	T8	T9
T1	×	←		←				>>	
T2		×		←					>>
T3			×	←			×		
T4				×	×	×			
T5					×	×			
T6						×		×	×
T7							×	←	
T8								×	×
T9									×

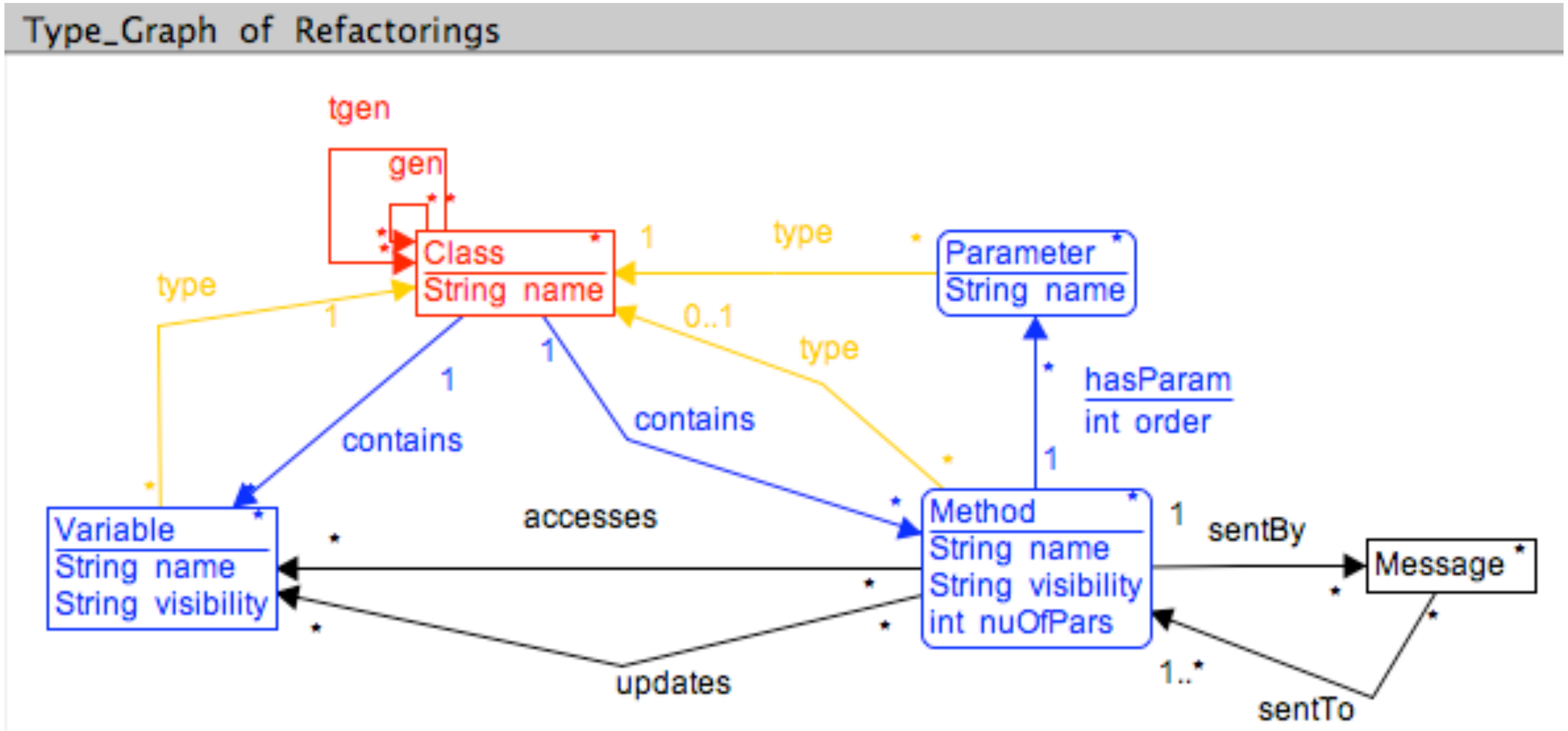
Applying graph transformation theory

- Approach: Use critical pair analysis in AGG
  - T_1 and T_2 form a *critical pair* if
    - they can both be applied to the same initial graph G but
    - applying T_1 prohibits application of T_2 and/or vice versa



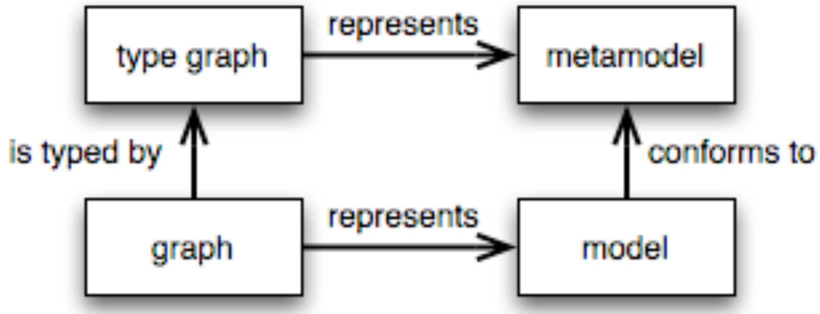
Applying graph transformation theory

Step 1: Express object-oriented metamodel as (attributed) type graph



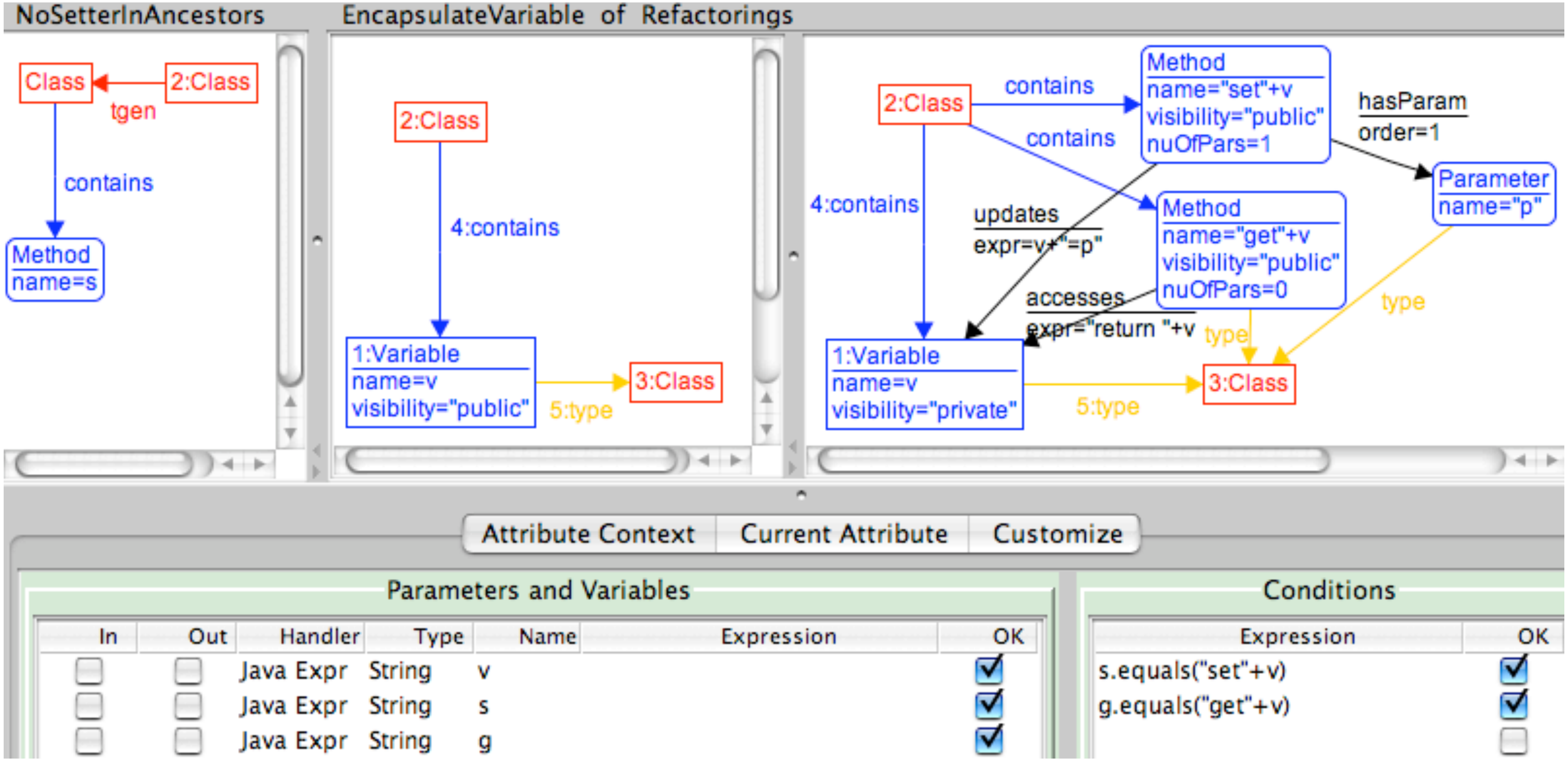
Interludium

- Type graphs versus metamodels



Applying graph transformation theory

Step 2: Express refactorings as (typed attributed) graph transformations



Applying graph transformation theory

Step 3: Detect critical pairs between refactoring transformations

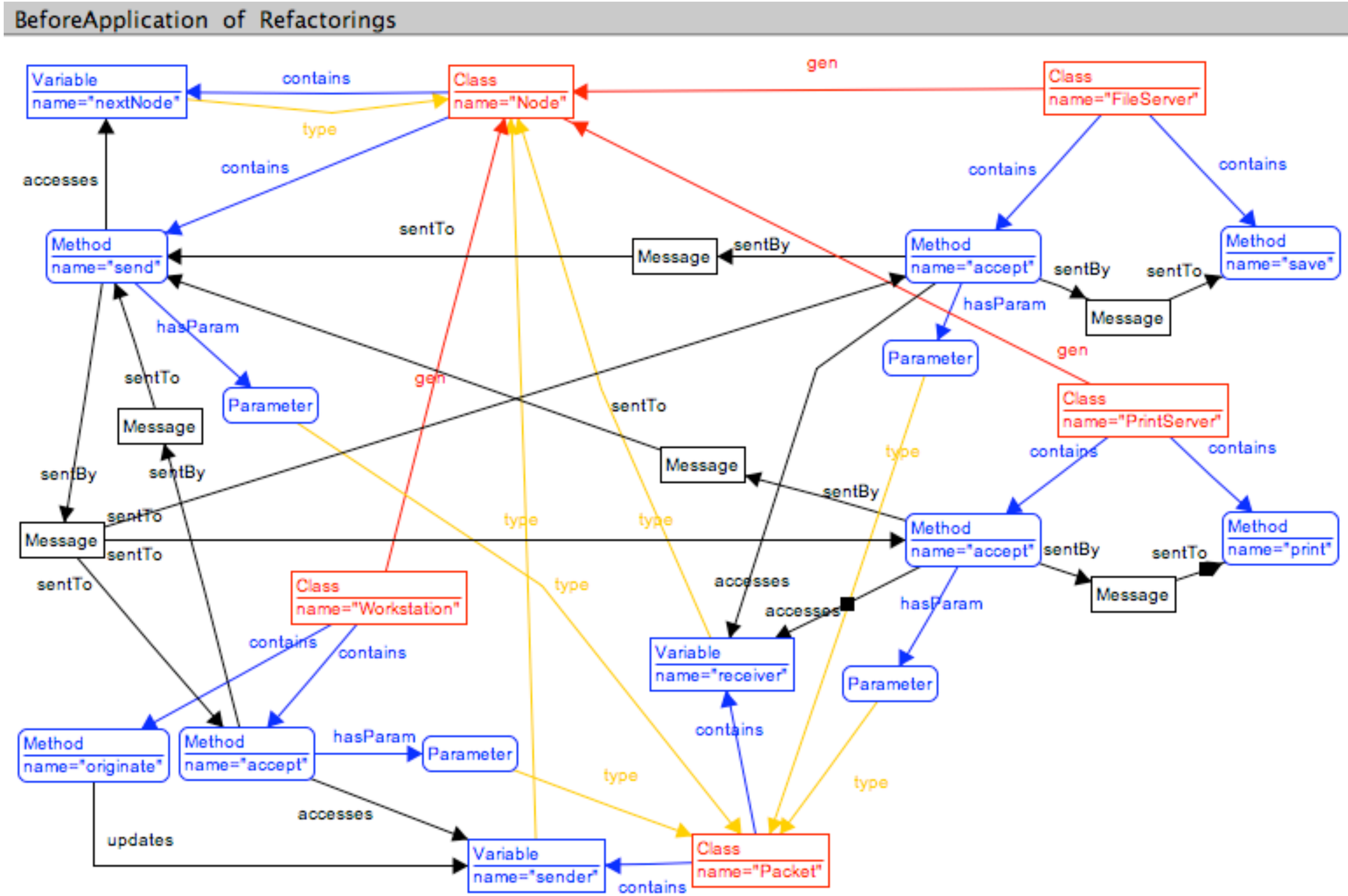
- *Potential conflicts* between refactorings

Critical Pairs

first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	2	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	2	0	0	0	0	0	0	1
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	1	1	1	0	0	2

Applying graph transformation theory

Step 4: Fine-tune critical pairs in context of concrete input graph



Applying graph transformation theory

Critical Pairs

first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	0	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	0	0	0	0	0	0	0	0
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0

Applying graph transformation theory

- Step 5: Perform sequential dependency analysis
  - To identify dependencies between refactorings that are applicable

Conclusion

- Graph transformation theory is a suitable formalism for understanding software refactoring

Graph Transformation	Refactoring
type graph, invariants	wf-constraints
negative application conditions	preconditions
parameterised graph production with NACs and context conditions mechanism	Refactoring transformation
Critical pair analysis	Detecting mutual exclusion
Confluence analysis	Detecting sequential dependencies