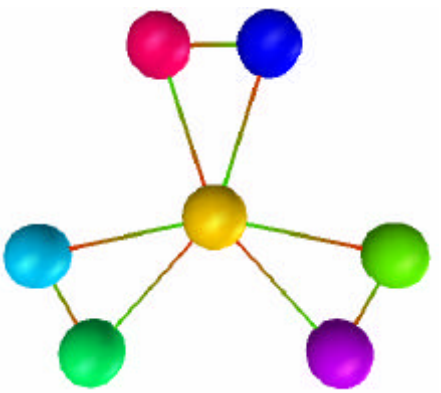
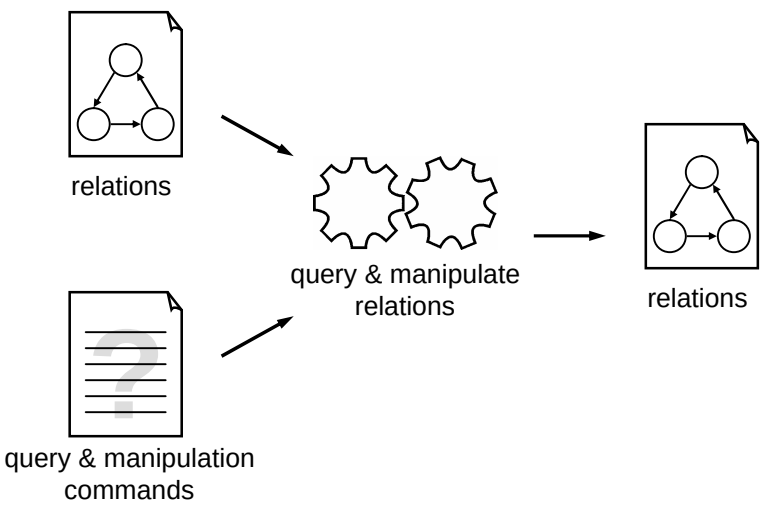
Simple and Efficient Relational Querying

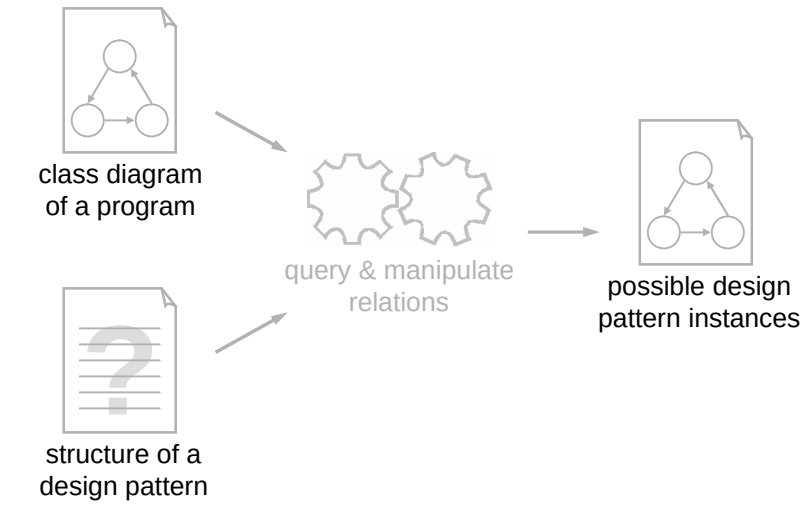
Dirk Beyer and Andreas Noack



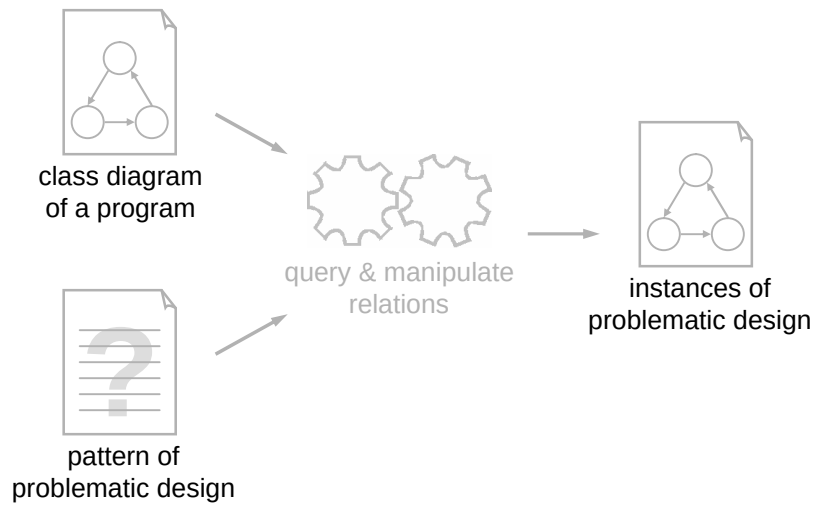
Querying and Manipulating Relations



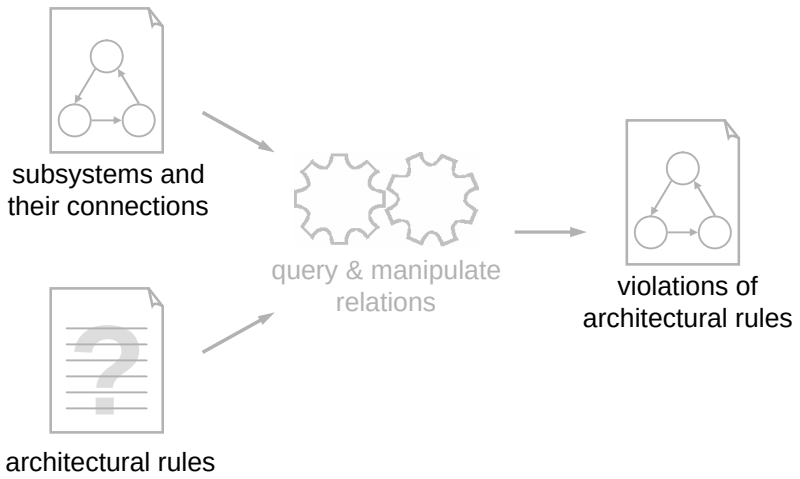
Applications in Reengineering (1)



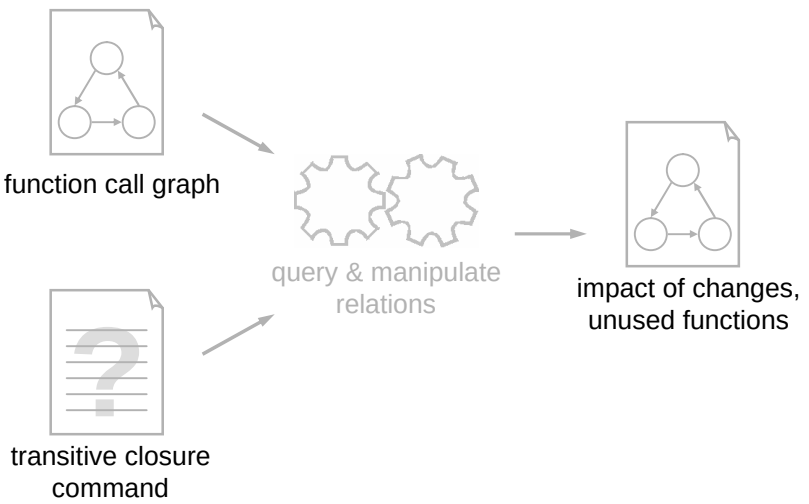
Applications in Reengineering (2)



Applications in Reengineering (3)



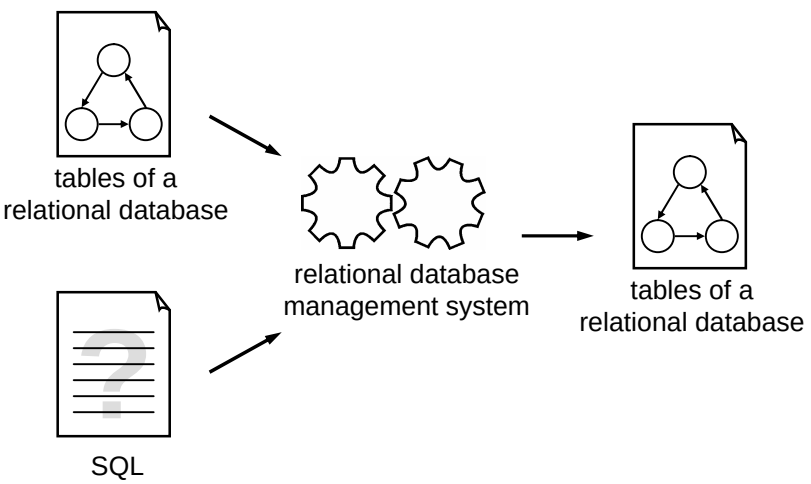
Applications in Reengineering (4)



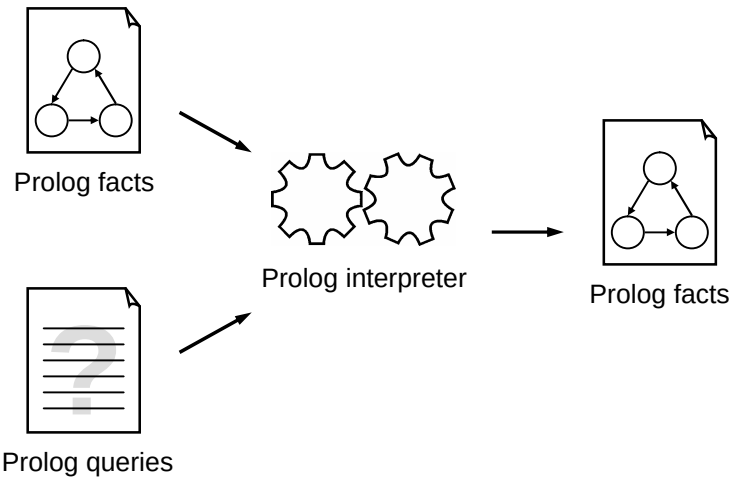
Previous Work

- relational databases and SQL
- Prolog
- calculators for relational algebra
  - Grok [Holt 1998]
  - RPA [Feijs et al. 1998]
  - RelView [Berghammer et al. 1998]
- textual and visual graph querying languages and tools
  - GReQL and GUPRO [Kullbach, Winter 1999]
  - GraphLog and Hy+ [Consens et al. 1992]

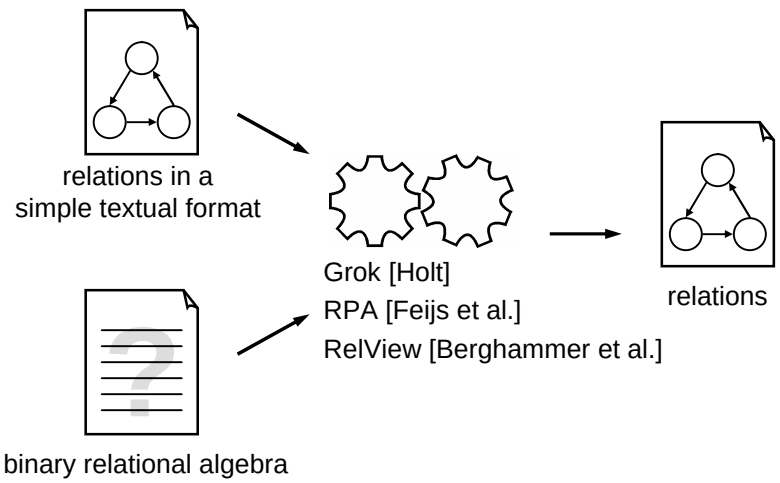
Related Work: RDMS and SQL



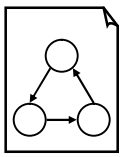
Related Work: Prolog



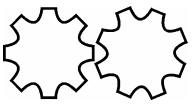
Related Work: Calculators for Binary Relations



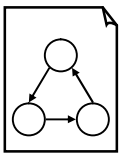
Our Approach: CrocoPat



input format for relations:
Rigi Standard Format



internal representation of relations:
binary decision diagram

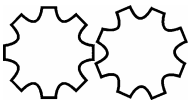


output format for relations:
Rigi Standard Format

query & manipulation language:
predicate calculus

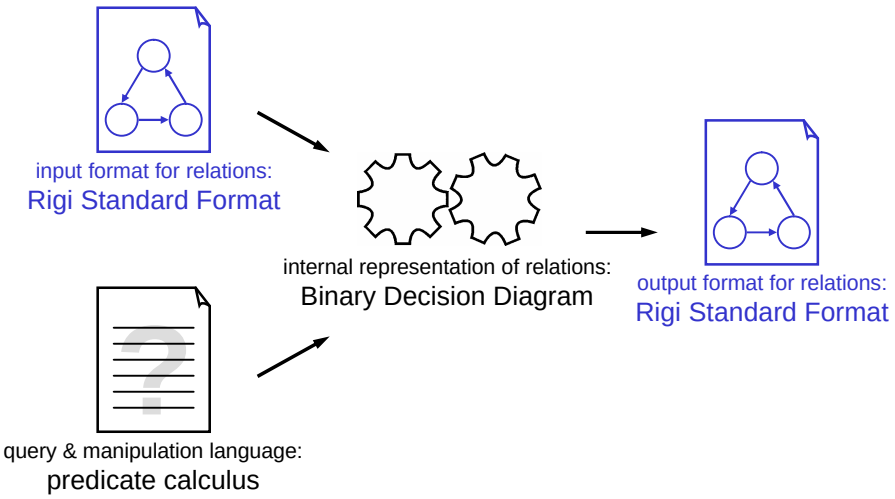
Contributions

Predicate calculus
as query and manipulation language
- expressive
  - not only binary relations (graphs), but *n*-ary relations
  → detection of graph patterns with more than two nodes
- well-known and fairly simple

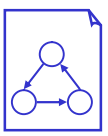


Binary decision diagrams (BDDs)
as data structure for relations
- efficient

Rigi Standard Format



Rigi Standard Format (RSF)



Java source code:

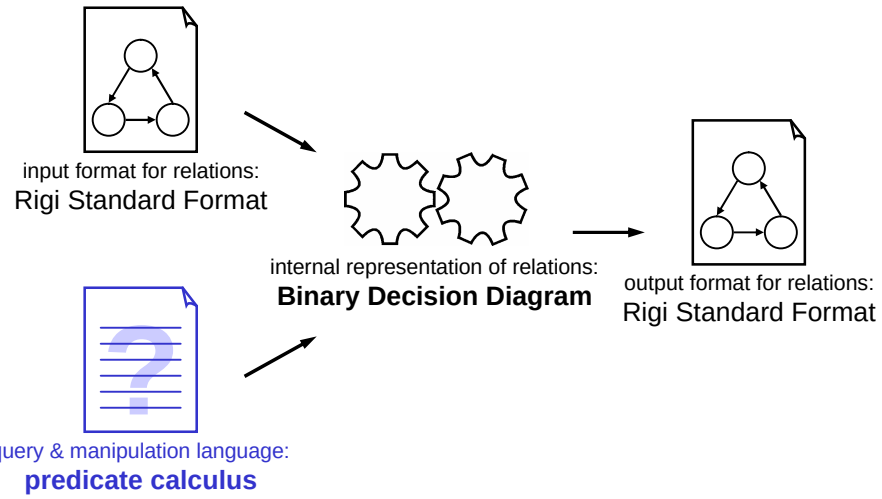
```
class ContainedClass {}
class SuperClass {}
class SubClass extends SuperClass {
   ContainedClass c;
}
```

extracted RSF file:

```
INHERIT SubClass SuperClass
CONTAIN SubClass ContainedClass
```

➔ simple
➔ facilitates data exchange with other tools

Query & Manipulation Language



Syntax

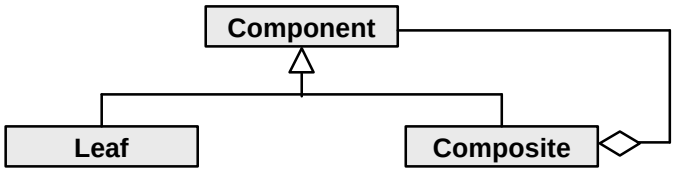
expression ::= (*expression*)
| TRUE
| FALSE
| *atomic_expression*
| *term* = *term*
| ! *expression*
| *expression* ^ *expression*
| *expression* + *expression*
| *expression* -> *expression*
| EX(*variable*, *expression*)
| FA(*variable*, *expression*)
| TC(*expression*, *variable*, *variable*)

atomic_expression ::= *relation_variable* (*term_list*)

term_list ::= *term* | *term_list*, *term*

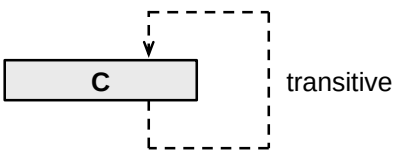
term ::= "*constant*" | *variable*

Example: Composite Design Pattern



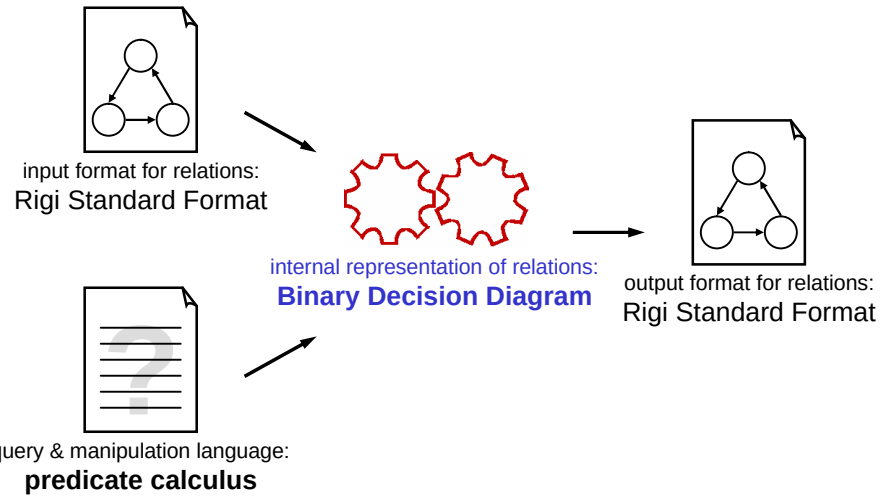

```
CompositePattern(Component, Composite, Leaf) :=
    INHERIT(Composite, Component)
  ^ CONTAIN(Composite, Component)
  ^ INHERIT(Leaf, Component)
  ^ ! CONTAIN(Leaf, Component);
```


Example: Cyclic Dependency




```
Depend(C1,C2) := CALL(C1,C2)
               + CONTAIN(C1,C2)
               + INHERIT(C1,C2);
DependTrans(C1,C2) := TC(Depend(C1,C2), C1, C2);
InCycle(C1) := EX(C2, DependTrans(C1,C2) ^ (C1=C2));
```

Internal Representation of Relations



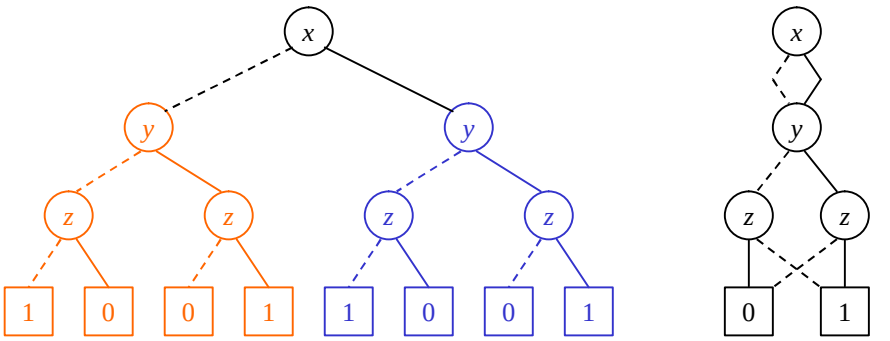
The Need for Efficiency

- Graph patterns with n nodes are n-tuples of nodes.
- For a graph with 1000 nodes, there are 1000^n n-tuples of nodes.

➔ To find graph patterns with more than 2 nodes,
huge relations have to be represented and manipulated.

The data structure **Binary Decision Diagram (BDD)** [Bryant 1986] can represent huge relations efficiently.

Binary Decision Diagram



- results from collapsing isomorphic subtrees of a decision tree

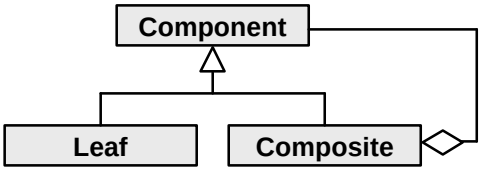
➔ redundancy is exploited to compress the representation

Experimental Evaluation: Systems

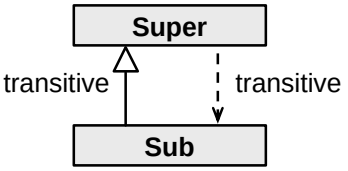
System	Classes	LOC	RSF Lines
JHotDraw 5.2	168	17 819	878
JDK 1.4.0 AWT	384	141 267	1 504
JWAM 1.6	2 397	284 818	12 298
JDK 1.4.0	5 312	1 179 576	28 699
Eclipse 2.02	8 925	1 181 270	63 121

- CrocoPat 2.1, MySQL 3.23.48 , Grok R15.0
- Linux PC with 1 GHz AMD Athlon CPU
- memory usage restricted to
  50 MB for CrocoPat,
  400 MB for MySQL and Grok

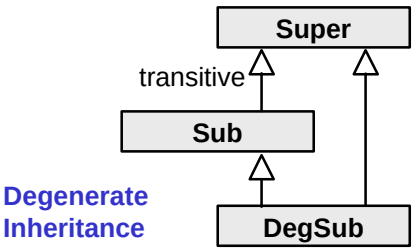
Experimental Evaluation: Queries



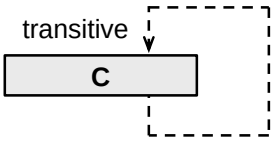
Composite Design Pattern



Subclass Knowledge



Degenerate Inheritance



Cyclic Dependency

Experimental Evaluation: Results

	JHotDraw	JDK AWT	JWAM	JDK	Eclipse
Composite (CrocoPat)	< 3 s			3.9 s	9.8 s
Deg. Inheritance (CrocoPat)				4.7 s	9.8 s

Experimental Evaluation: Results

	JHotDraw	JDK AWT	JWAM	JDK	Eclipse
Subcl. Knowledge (Grok)	< 3 s			75.6 s	>400 MB
Subcl. Knowledge (CrocoPat)				8.3 s	30.3 s
Cyclic Dependency (Grok)				115 s	>400 MB
Cyclic Dependency (CrocoPat)				8.4 s	31.9 s

Experimental Evaluation: MySQL vs. CrocoPat

- transitive closure of the union of `CALL`, `CONTAIN`, and `INHERIT`

	JHotDraw	JDK AWT	JWAM	JDK	Eclipse
Transitive Closure (MySQL)	0.35 s	29.5 s	16.8 s	>400 MB or >1 h	>400 MB or >1 h
Transitive Closure (CrocoPat)	0.18 s	0.27 s	2.24 s	8.37 s	30.9 s

Applications

- completed: integration with Sotograph,
  a commercial software analysis workbench

- ongoing: integration with visualization

- for your application, CrocoPat 2.1:
  - is Open Source
  - has Quality
  - is available at:

    http://www.google.com/search?q=**CrocoPat**

Conclusions

The predicate calculus based language
- is fairly easy to use and
- sufficiently expressive for many structural software analyses,
  including the detection of graph patterns.

The BDD based implementation
- scales well to the analysis of large systems.

The tool CrocoPat
- is easy to integrate with other tools.
- is applied in industry.