Applying Formal Concept Analysis to Reverse Engineer Software

Jukka Viljamaa

PhD
assistant
Department of Computer Science
University of Helsinki

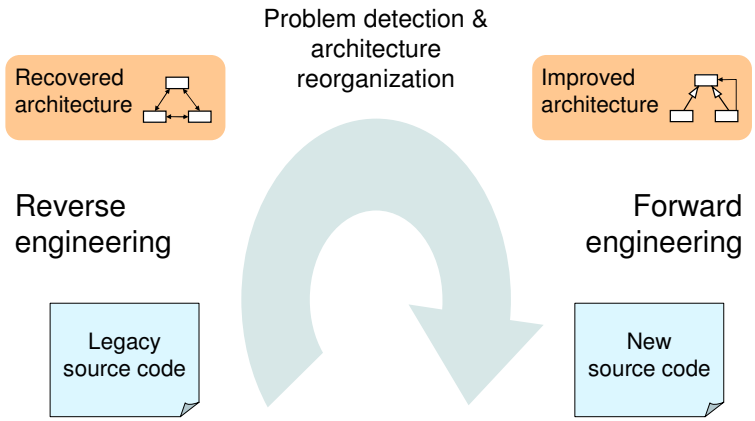
Presentation Outline

- Introduction
  - reverse engineering
  - formal concept analysis (FCA)
- Examples of Applying FCA
  - analyzing the impact of change
  - reverse engineering framework reuse interfaces
  - mapping user-triggered features to implementation
- Conclusions

Reverse Engineering

- Reverse engineering as part of *reengineering* process
  - R.E. is "*the process of deriving abstract formal specifications from the source code of a legacy system, where these specifications can be used to forward engineer a new implementation of that system.*" *[Arnold, 1992]*
- More generally, a three-step process to
  - extract high-level descriptions from system implementation
    (1) information gathering
    (2) organization
    (3) presentation, navigation & analysis *[Tilley, 1995]*

RE in Reengineering Process



Approaches & Applications

- ***Matching (libraries of) structural templates (patterns)***
  - system architecture recovery *[Harris et al., 1995; Keller et al., 1999]*
  - design pattern recovery *[Brown, 1996; Seemann & Gudenberg, 1998; Ferenc et al., 2001; Niere et al., 2002]*
- ***Cluster analysis*** based on, e.g., similarity metrics
  - automatic remodularization *[Schwanke, 1991]*
  - system structuring at file level *[Mancoridis et al., 1999; Tzerpos & Holt, 2000; Maletic & Marcus, 2001]*
  - system architecture recovery *[Bauer & Trifu, 2000]*
  - object identification in procedural code *[Quigley et al., 2000]*
  - visualization *[Demeyer et al., 1999; Lanza & Ducasse, 2001]*

Approaches & Applications

- ***Formal concept analysis*** *[Ganter & Wille, 1999]*
  - analyzing source code configurations *[Krone & Snelting, 1994]*
  - global variable usage in procedures *[Lindig & Snelting, 1997]*
  - reengineering class hierarchies *[Snelting & Tip, 1998]*
  - semi-automatic software modularization *[Siff & Reps, 1997; Tonella, 2001; Al-Ekram & Kontogiannis, 2004]*
  - design pattern recovery *[Tonella & Antoniol, 1999]*
- …

Formal Concept Analysis

- Discovering sensible groupings of ***objects*** that have common ***attributes*** in a certain ***context***

$$C = (O, A, I)$$

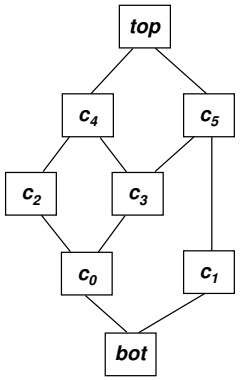
- ***Concept*** is a maximal set of objects (***extent***) sharing a set of attributes (***intent***)

$$(X \subseteq O,\ Y \subseteq A) \text{ so that}$$
$$X = \tau(Y) = \{o \in O \mid \forall a \in Y\colon (o, a) \in I\} \text{ and}$$
$$Y = \sigma(X) = \{a \in A \mid \forall o \in X\colon (o, a) \in I\}$$

- There are algorithmic ways of producing a ***concept lattice*** showing all the concepts and their relationships from any context

$$\sup_{i \in I}(X_i, Y_i) = (\tau(\sigma(\bigcup_{i \in I} X_i)), \bigcap_{i \in I} Y_i) = (\tau(\bigcap_{i \in I} Y_i), \bigcap_{i \in I} Y_i)$$

Constructing Concept Lattice



	athletics	ballgame	team sp.	Olympic sp.
volleyball		√	√	√
javelin	√			√
long jump	√			√
cricket		√	√	
tennis		√		√
bowling		√		

top: ({bowling, cricket, javelin, long jump, tennis, volleyball}, ∅)
c_5: ({tennis, volleyball, javelin, long jump}, {Olympic sport})
c_4: ({bowling, cricket, tennis, volleyball}, {ballgame})
c_3: ({tennis, volleyball}, {ballgame, Olympic sport})
c_2: ({cricket, volleyball}, {ballgame, team sport})
c_1: ({javelin, long jump}, {athletics, Olympic sport})
c_0: ({volleyball}, {ballgame, team sport, Olympic sport})
bot: (∅, {athletics, ballgame, team sport, Olympic sport})

Three Examples of Applying FCA

1. **Change impact analysis** *[Tonella, 2003]*
   - "Which other parts of the implementation are affected if I change this line of code?"
2. **Reuse interface analysis** *[Viljamaa, 2003]*
   - "How do I (re)use this framework or platform?"
   - "Can I extract reuse policy specifications from existing successful examples?"
3. **Feature analysis** *[Eisenbarth, Koschke & Simon, 2003]*
   - "Where is this feature implemented in code?"

Example 1: Change Impact Analysis *[Tonella, 2003]*

- Extension of traditional slicing: ***concept lattice of decomposition slices***
  - shows all static dependencies between the computations of a program
  - shows also ***weak interferences***
    - i.e. sets of shared statements that affect the values of some variables, but which are not the only statements that contribute to their values

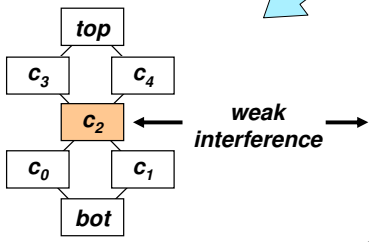
Change Impact Analysis: Setting

- FCA objects: program variables
- FCA attributes: program statements
- FCA context relation:
  - contains a pair (*v*, *s*) if statement *s* belongs to the decomposition slice of variable *v*
- Resulting concepts contain
  - sets of variables (extent)
  - that share some computation (intent)
- Produced concept lattice
  - shows the hierarchical dependencies of the computations
  - answers questions like: "Does a change of the computation on variable *x* affect the computation performed on variable *y*?"

Change Impact Analysis: Example

```
 1:  scanf("%d", &f);
 2:  scanf("%d", &g);
 3:  scanf("%d", &c);
 4:  scanf("%d", &d);
 5:  if (f > 5) f++;
 6:  if (g > 5) g++;
 7:  if (c > 5) c++;
 8:  if (d > 5) d++;
 9:  f = f + c + d;
10:  g = g + c + d;
```

	1	2	3	4	5	6	7	8	9	10
c			√				√			
d				√				√		
f	√		√	√	√		√	√	√	
g		√	√	√		√	√	√		√

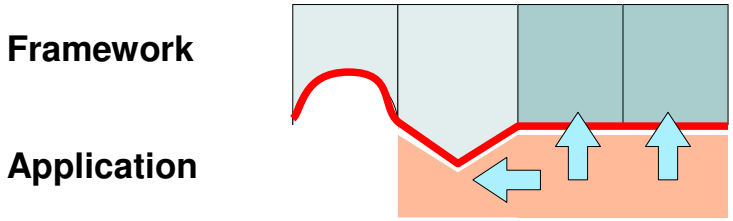


top	({*c*, *d*, *f*, *g*}, ∅)
c_4	({*d*, *f*, *g*}, {*4*, *8*})
c_3	({*c*, *f*, *g*}, {*3*, *7*})
c_2	({*f*, *g*}, {*3*, *4*, *7*, *8*})
c_1	({*g*}, {*2*, *3*, *4*, *6*, *7*, *8*, *10*})
c_0	({*f*}, {*1*, *3*, *4*, *5*, *7*, *8*, *9*})
bot	(∅, {*1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*})

Change Impact Analysis: Results

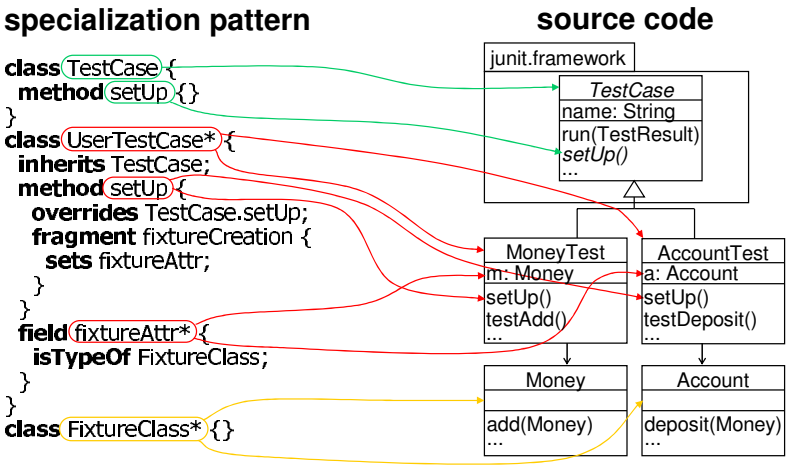
- Helps assessing the consequences of a change
  - find the other statements that are affected
  - the size of the impact zone provides an objective, quantitative indicator of the difficulty of the change
- Helps regression testing
  - no need to retest the code not in the impact zone

Example 2: Assisting Framework-Based Development *[Viljamaa, 2003]*

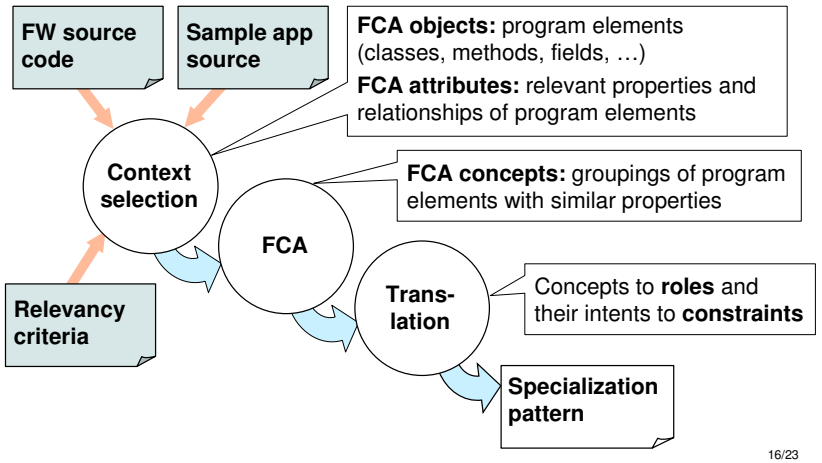


reuse interface = a collection of hot spots or variation points that enable specialization of framework, e.g. by subclassing (*white-box reuse*) or by combining and customizing ready-made components (*black-box reuse*)

Specialization Patterns



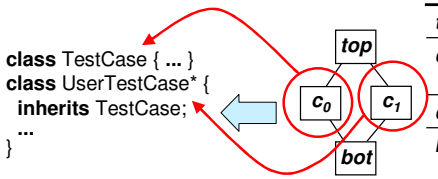
Reuse Interface Analysis: Process



Reuse Interface Analysis: Example

```
class TestCase {
  void setUp() {}
}
class MoneyTest
  extends TestCase {
  void setUp() { … }
}
class AccountTest
  extends TestCase {
  void setUp() { … }
}
```

	name TestCase	*inherits TestCase*	*overrides setUp*
TestCase	√		
MoneyTest		√	√
AccountTest		√	√



top	({TestCase, AccountTest, MoneyTest}, ∅)
c_1	({MoneyTest, AccountTest}, {inherits TestCase, overrides setUp})
c_0	({TestCase}, {name TestCase})
bot	(∅, {name TestCase, inherits TestCase, overrides setUp})

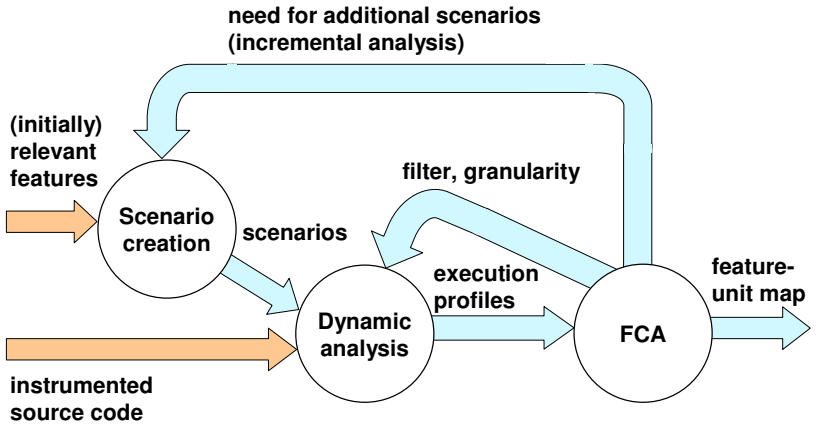
Reuse Interface Analysis: Results

- FCA enables automatic extraction of over 50 % of framework reuse interface specification
- FCA-based automation
  - yields an accurate overall picture of the patterns
  - makes reuse interface modeling faster
  - allows modeling of frameworks under development
  - can increase the quality of the produced models

Example 3: Mapping Features to Implementation *[Eisenbarth, Koschke & Simon, 2003]*

- *Static analysis* to identify the *computational units*
  - e.g. functions or methods
- *Scenario creation*, *instrumentation* and *dynamic analysis* to gather *execution profiles*
  - i.e. which units get called on which execution
- FCA to build a map that shows the relationships between the features and the implementation
  - FCA objects: the computational units
  - FCA attributes: the scenarios created by the analyst
  - FCA context: the execution profiles

Feature Analysis: Process



Feature Analysis: Results

- Using the resulting concept lattice the analyst can answer questions such as
  - “Which are the computational units required for all scenarios?” (the bottom concept’s extent)
  - “Is a computational unit *u* specific to exactly one scenario *s*?” (true if *s* is the only scenario on all paths from the lowest-level concept whose extent contains *u* to the top concept)

Conclusions — Pros of FCA

- Solid theoretical foundation
- Generality allows wide range of opportunities for applying the method (lots of examples available)
- Effectiveness demonstrated in practice
- Already (some) tool-support available
- No need for templates or pattern matching

Conclusions — Cons of FCA

- Performance
  - worst case computational complexity exponential
  - in practice scalability reasonable if focused analysis & optimized algorithms are used
- Generality
  - how to apply?
    - new ways of applying need to be carefully designed and implemented (may require experimentation)
  - interpretation of the results? (no universal guidelines)