15 Exhaustive Graph Rewrite Systems (XGRS) for Refactorings and Other Transformations

Prof. Dr. Uwe Aßmann
Softwaretechnologie
Technische Universität Dresden
Version 15-0.4, 02.01.16

1) Termination of EARS
2) Termination of AGRS
3) SGRS
4) XGRS
5) Refactoring Example
Obligatory Literature

  - http://portal.acm.org/citation.cfm?id=363914


- Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/
Remember: Rename Refactorings in Programs

Refactor the name Person to Human, using bidirectional use-def-use links:

```java
class Person {
    // ..
}
class Course {
    Person teacher = new Person("Jim");
    Person student = new Person("John");
}
```

```java
class Human {
    // ..
}
class Course {
    Human teacher = new Human("Jim");
    Human student = new Human("John");
}
```
Refactoring as Graph Transformation

- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions

- Refactoring can be supported by Graph Rewrite Tools
  - The Use-Def-Use-graph (UDUG) forms the basis of refactoring tools
  - Build up the UDUG with graph analysis (EARS)
  - Rewrite it with graph rewriting (XGRS)
15.1 Termination and Confluence of EARS

A Fujaba GRS (in one activity of the storyboard) may terminate and deliver a unique result.
Problems with GRS

With graph rewriting for model and program analysis, refactoring, and transformation, there are some problems:

- **Termination**: The rules of a GRS G are applied in chaotic order to the manipulated graph. When does G terminate for a start graph?
  - Idea: can we „forcefully“ terminate the rewriting?
  - Idea: identify a termination graph which stops the rewriting when completed

- **Non-convergence (indeterminism)**: when does a GRS deliver a deterministic solution (unique normal form)?
  - Can we automatically select a “standard” normal form?
  - Idea: unique normal forms by rule stratification
Additive Termination

• A termination subgraph is a subgraph of the manipulated graph, which is step by step completed.
• Conditions in the additive case:
  – nodes of termination (sub-)graph are not added (remain unchanged)
  – its edges are only added
• If the termination graph is complete, the system terminates.
Transitivising (Flattening) the Inheritance Hierarchy

- Does this rule terminate?
  - Yes, because EARS complete graphs and shorten paths
  - “is-a” is the termination subgraph

- Fujaba GRS rule “FlattenInheritanceHierarchy”:

```
<<create>>
A:Class
B:Class
C:Class
FlattenInheritanceHierarchy()
```

[Christoph04]
Run-Time Derivation (Snapshots): Transitivitying the Inheritance Hierarchy

- Ex.: A simple class inheritance tree (acyclic) is “shortened”
- “is-a” is completed step by step
Transitivising the Inheritance Hierarchy

- Ex.: A simple class inheritance tree (acyclic) is “shortened”
Transitivising the Inheritance Hierarchy

- If every indirect path is shortened, rewriting stops
Example: Collect Subexpressions

- EARS also work on bipartite graphs

- Query to build up the use-definition-use graph (UDUG) between Statements and Expressions:
  - "Find all subexpressions which are reachable from a statement"

/// F-Datalog:
ExprsOfStmt(Stmt:Statement,Expr:Expr) :-
  Child(Stmt,Expr2), Descendant(Expr2,Expr).

// Descendant is transitive closure of Child
Descendant(Expr1:Expr,Expr2:Expr) :- Child(Expr1,Expr2).
Descendant(Expr1:Expr,Expr2:Expr) :- Descendant(Expr1,Expr3),
  Child(Expr3,Expr2).

- Features of graph rewrite system:
  - terminating, strong confluent
  - convergent (unique normal form)
  - recursive
EARS CollectExpressions

- Two transitive closures, specified as path abbreviations

![Diagram showing the process of collecting expressions with transitive closures.]
Execution of „Reachable Subexpressions“

- Start situation

```
Assign
   Child
   Expr
      Const
         1
   Child
   Plus
      Var
         X
```
Execution of „Reachable Subexpressions“

- Why do such graph rewrite systems terminate? Answer: `ExprsOfStmt` and `Descendants` are termination subgraphs, completed step by step.

```
Assign
  ├── Child
  │    └── Expr
  │         ├── Const
  │         │    └── 1
  │         └── Var
  │              └── X
  │                  └── Descendants
  └── ExprsOfStmt
        └── Descendants
```

Descendants
Execution of „Reachable Subexpressions“

Diagram:
- Assign
  - Child
  - Expr
    - Const
      - 1
    - Var
      - X
- Plus
  - ExprOfStmt
  - Descendants
- Descendants
Execution of „Reachable Subexpressions“

Diagram showing the execution of reachable subexpressions with nodes for Assign, Expr, Plus, Const, and Var. Edges indicate relationships such as Child, ExprsOfStmt, and Descendants.
Execution of „Reachable Subexpressions“
EARS - Simple Edge-Additive GRS

- **EARS (Edge addition rewrite systems)** only add edges to graphs
  - They can be used for the construction of graphs
  - For the building up analysis information about a program or a model
  - For abstract interpretation on an abstract domain represented by a graph
- **terminating**: terminating on the finite lattice of subgraphs of the manipulated graph
  - Added edges form the termination subgraph
- **strongly confluent**: direct derivations can always be interchanged.
- **congruent**: unique normal form (result)
- ==> If a Fujaba activity contains an EARS, it terminates and delivers a unique result
Name and Type Analysis with EARS

- EARS are very useful for program analysis problems
- Uses of names must be linked to their definitions
  - procedures, methods
  - classes, types
- Name analysis looks up used names in the context
  - Search
  - Lookup in tables
  - Reachability analysis: if a definition of a used name is reachable, then it forms a use-def edge in the use-def graph
Data-flow Analysis with EARS

- EARS are very useful for program analysis problems
- Every distributive data-flow problem (abstract interpretation problem) on finite-height powerset lattices can be represented by an EARS
  - defined/used-data-flow analysis
  - partial redundancies
  - local analysis and preprocessing:

- EARS are equivalent to binary F-Datalog
- EARS work for other analysis problems, which can be expressed with F-Datalog-queries
  - equivalence classes on objects
  - alias analysis
  - program flow analysis
15.2 Termination of Additive GRS (AGRS)

- Sometimes, during refactoring and transformations, we must allow for node additions, nodes which should represent new information.
Example: Allocation of Register Objects for Storing the Result of Expressions in Statements

- Query: "Allocate a register object for every subexpression of a statement which has a result and link the expression to the statement"

```java
if ExprsOfStmt(Stmt, Expr), HasResult[Expr]
then
  ObjectExprs(Stmt, Expr),
  RegisterObject := new Register;
  UsedReg(Expr, RegisterObject)
;
- Features: terminating
```
- ObjectExprs is the “termination subgraph”, i.e., the subgraph which cannot grow out of bound
- **ObjectExprs** is the “termination subgraph”, i.e., the subgraph which cannot grow out of bound.
A Derivation with the Termination Subgraph Will Stop

[Àßmann00]
Edge-Accumulative Rules and AGRS

- A GRS is called **edge-accumulative (an AGRS)** if
  - all rules are edge-accumulative and
  - no rule adds nodes to the termination-subgraph nodes of another rule.
- Edge-accumulative rules are defined on label sets of nodes and edges in rules
- This criterion statically decidable
The Termination Subgraph of the Examples

Collection of subexpressions:
T = ([Stmt,Expr], {ExprsOfStmt, Descendant} )

Allocation of register objects:
T = ([Proc,Expr], {ObjectExprs} )
15.3 Subtractive GRS (SGRS)
Subtractive Termination

- **Conditions in the subtractive case:**
  - the nodes of the termination subgraph are not added (remain unchanged)
  - its edges are only deleted
- **If the termination subgraph is empty, the system terminates**
- **Results in:**
  - edge-subtractive GRS (ESGRS)
  - subtractive GRS (SGRS)
Constant Folding as Graph Rewrite Rule

Graph representation:
- **Plus**
  - Input: **Const** (1)
  - Input: **Const** (2)
- **Const** (3)

Diagram:
- **Plus** node connected to two **Const** nodes (1 and 2).
- Resulting **Const** node (3).

Explanation:
- The graph rewrite rule demonstrates constant folding, where two constant operations are combined into a single constant operation.
- The rule transforms the expression `const 1 + const 2` into `const 3`.
Peephole Optimization as Subtractive XGRS

![Diagram of Peephole Optimization as Subtractive XGRS]

- **Plus**
  - **Var** → **X**
  - **Const** → **1**
  - Next

- **Incr**
  - **X**

- **IncrIncr**
  - **Var** → **X**
15.4 Exhaustive GRS (XGRS)
AGRS and SGRS make up *exhaustive graph rewriting systems (XGRS)*

In an XGRs, all redexes in the termination-subgraph are consumed step by step.

- The termination-subgraph is either *completed* or *consumed*
  - Edge-accumulative systems may create new redex parts in the termination-subgraph, but
    - there will be at most as many of them as the number of edges in the termination-subgraph.
  - Subtractive systems do not create sub-redexes in the termination-subgraph but destroy them.

- XGRS can only be used to specify algorithms which
  - perform a *finite* number of actions depending on the size of the host graph.
15.5 Refactoring Example “Pull-Up Features into Common Superclass”

[Christoph04]
Step 1: Flattening the Inheritance Hierarchy

- This rule terminates, due to path contraction and subtraction
- The rule, FlattenClassHierarchy, has a unique normal form
Step 2: 
Pull-Up-Method Refactoring

- Additive Step: Create a new base class for common attributes; mark this as the new “base-type” of the attribute

- The rule, Mark-Pull-Up, has a unique normal form

```
NewSuper.Name := "<A.name>_B.name>_Base"
```

Diagram:
- A:Class
- B:Class
- C:Attribute

Mark-Pull-Up

NewSuper:Class

Contains:
- <<create>>
- <<create>>
- base-type

Contains:
- <<create>>
Step 3

- Edge-Additive Step: alternate case: a class A has attributes that should be moved up anyway

- The rule, Mark-Pull-Up-2, has a unique normal form
Step 4

- Subtractive Step: do the real "pull-up" into the superclass
- The rule, Pull-Up-Features, has a unique normal form
Putting it All Together

- The rule sequence
- \{ FlattenClassHierarchy, Mark-Pull-Up, Mark-Pull-Up-2, Pull-Up-Features \}
- is terminating (XGRS) and confluent
- has a unique result, the desired refactored class hierarchy

- We specified a refactoring with only 4 rules
The End

- Many model and program transformations can be specified by terminating XGRS
- Termination criteria build on a *termination subgraph* that is completed or deleted during the transformation
- Refactorings on the UDUG can be described with graph transformations
- Fujaba storyboards allow for chaining XGRS, so that the overall chain terminates