14b-ST2, 37-SEW
Exhaustive Graph Rewrite Systems (XGRS) for Model and Program Transformations

Prof. Dr. Uwe Aßmann

Softwaretechnologie

Technische Universität Dresden

Version 11-0.4, 10.12.11

1) EARS
2) AGRS
3) SGRS
4) XGRS

SEW, © Prof. Uwe Aßmann

Obligatory Literature

▲ Uwe Aßmann. Graph rewrite systems for program optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 22(4):583-637, June 2000.

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

▲ Alexander Christoph. Graph rewrite systems for software design transformations. In M. Aksit, editor, Proceedings of Net Object Days 2002, Erfurt, Germany, October 2002.

▲ Alexander Christoph. GREAT - a graph rewriting transformation framework for designs. Electronic Notes in Theoretical Computer Science (ENTCS), 82 (4), April 2003.

▲ Alexander Christoph. Describing horizontal model transformations with graph rewriting rules. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, MDAFA, volume 3599 of Lecture Notes in Computer Science, pages 93-107. Springer, 2004.

▲ Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143

  ▪ http://www.springerlink.com/content/574224611510743l/

Prof. U. Aßmann, SEW

36.1 EARS

Problems with GRS
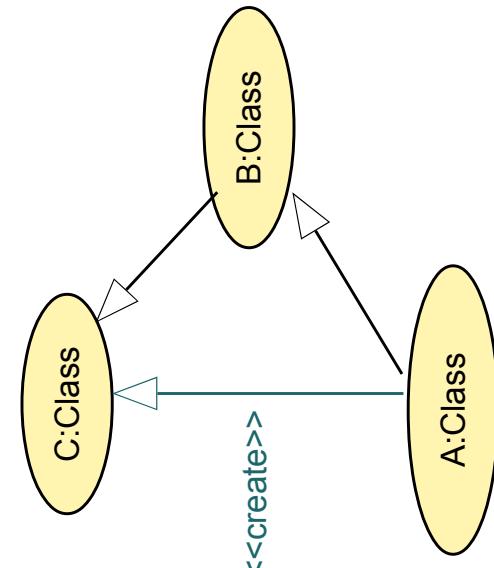
With graph rewriting for model and program transformation, there are some problems:

▲ **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: identify a *termination graph* which stops the rewriting when completed

▲ **Non-convergence (indeterminism)**: when does a GRS deliver a deterministic solution (unique 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 the Inheritance Hierarchy

[Christoph04]

Example: Collect Subexpressions

- "Find all subexpressions which are reachable from a statement"

ExprsOfStmt[Stmt,Expr] :- Child[Stmt,Expr].
ExprsOfStmt[Stmt,Expr] :- Child[Stmt,Expr2], Descendant[Expr2,Expr].
// Descendant is transitive closure of Child
Descendant[Expr1,Expr2] :- Child[Expr1,Expr2].
Descendant[Expr1,Expr2] :- Descendant[Expr1,Expr3],
                Child[Expr3,Expr2].

- Features:
  - terminating, strong confluent
  - convergent (unique normal form)
  - recursive
- Why do such graph rewrite systems terminate?
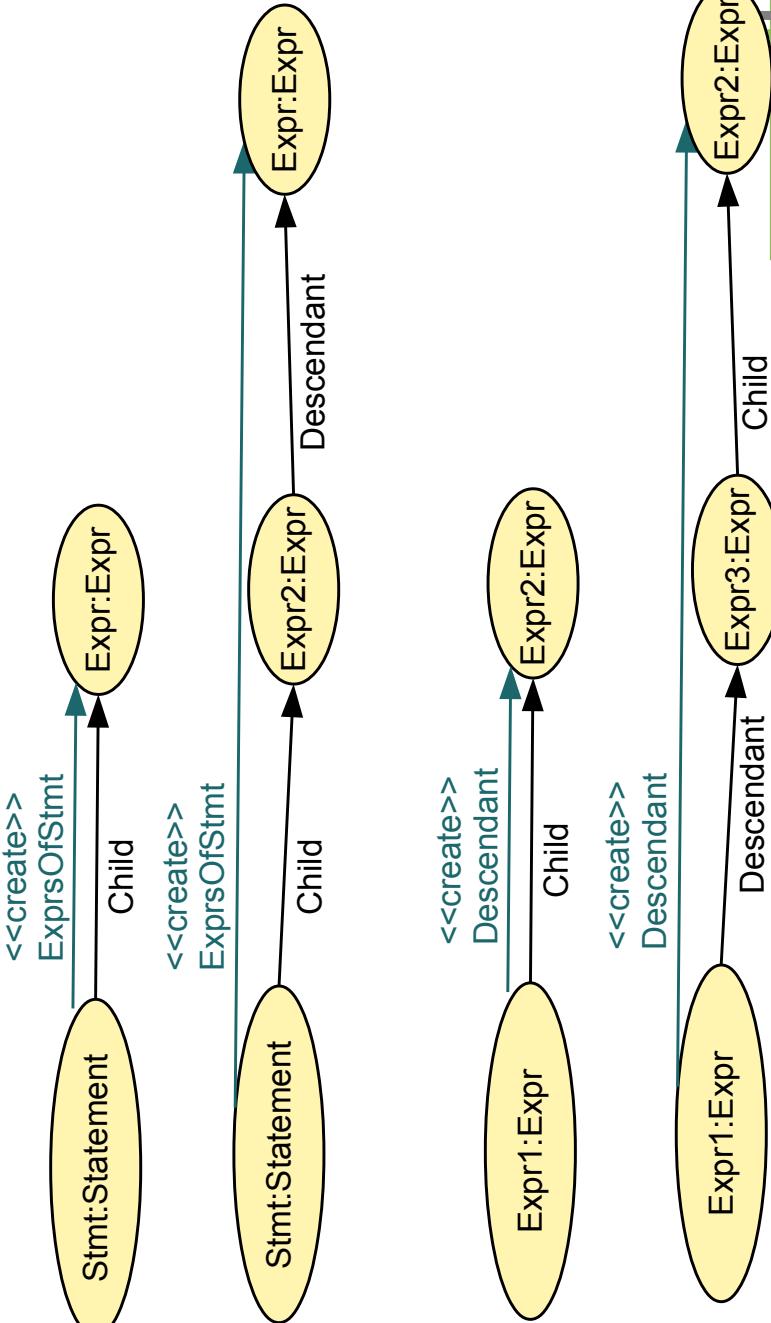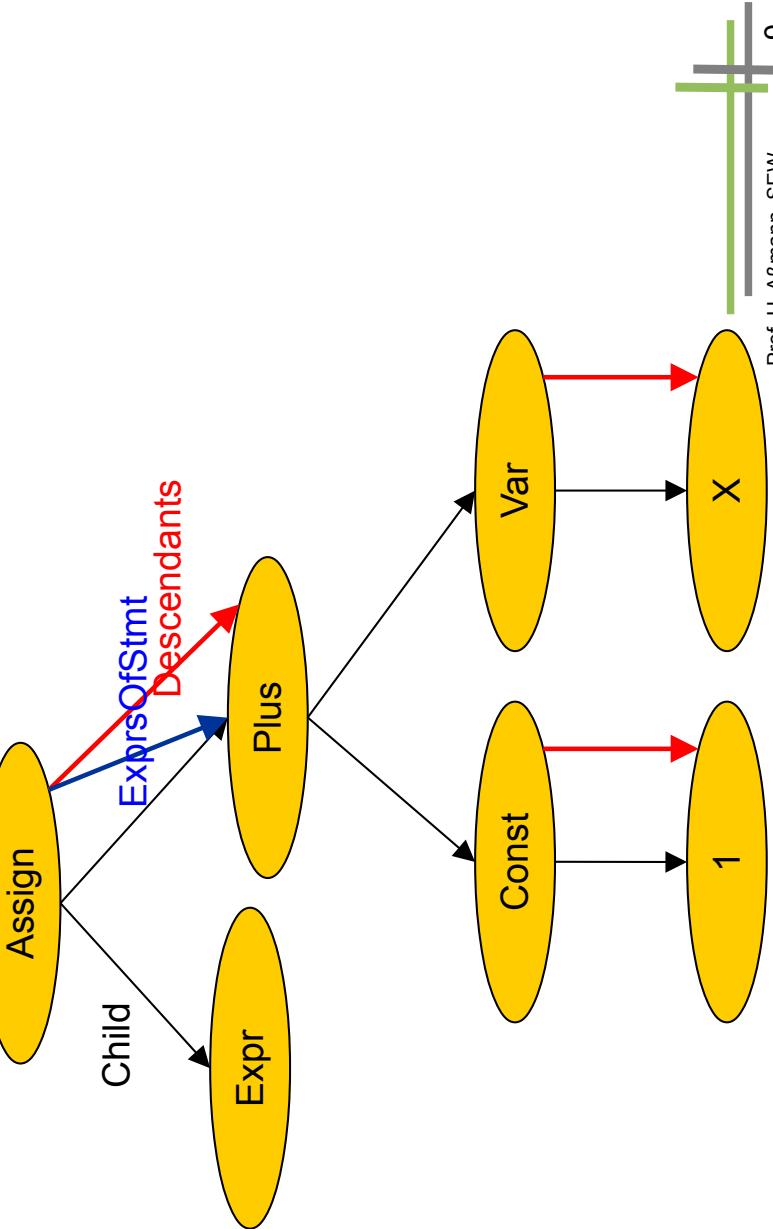
EARS CollectExpressions

- Two transitive closures
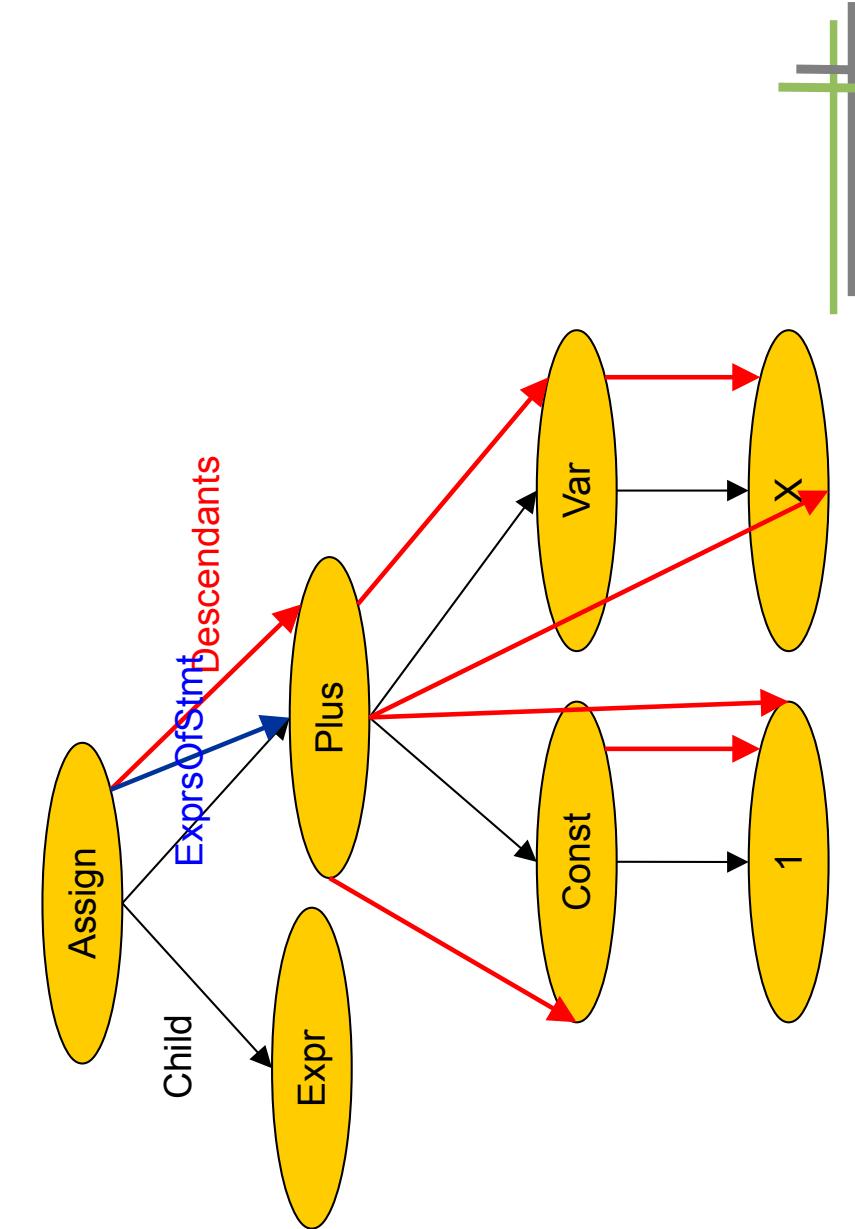
Execution of „Reachable Subexpressions"
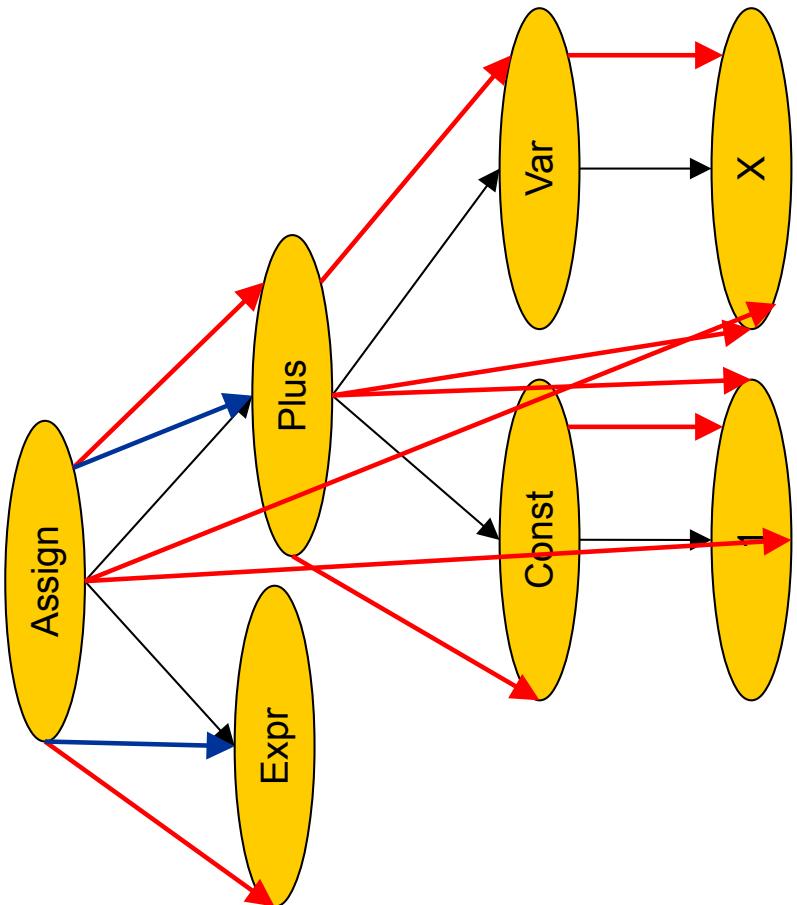
- Answer: ExprsOfStmt and Descendants are termination subgraphs, completed step by step
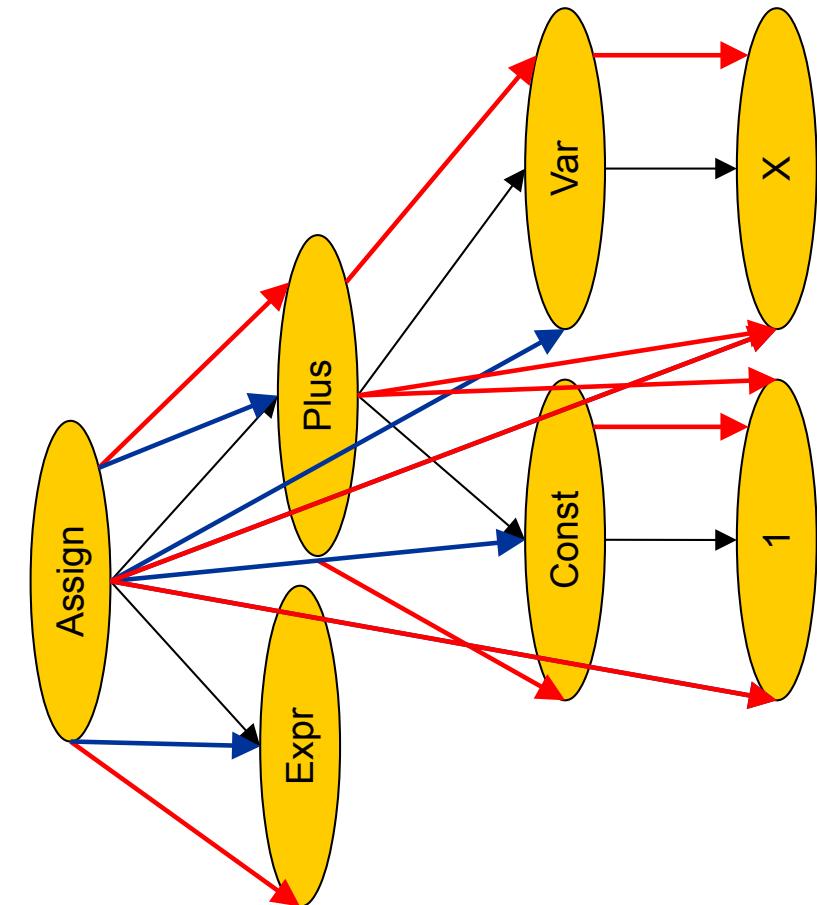
Assign

Child

Expr

ExprsOfStmt

Descendants

Plus

Var

Const

X

1

Execution of „Reachable Subexpressions"

Assign

Child

Expr

ExprsOfStmt

Descendants

Plus

Var

Const

X

1

Execution of „Reachable Subexpressions"

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**: noetherian 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)
- EARS are equivalent to binary F-Datalog

Data-flow Analysis with EARS

- 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 work for other problems which can be expressed with DATALOG-queries
  - equivalence classes on objects
 - alias analysis
  - program flow analysis

36.2 Additive GRS (AGRS)
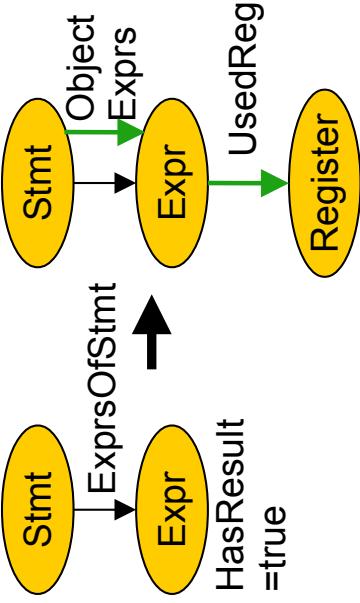
Example: Allocation of Register Objects

- "Allocate a register object for every subexpression of a statement which has a result and link the expression to the statement"
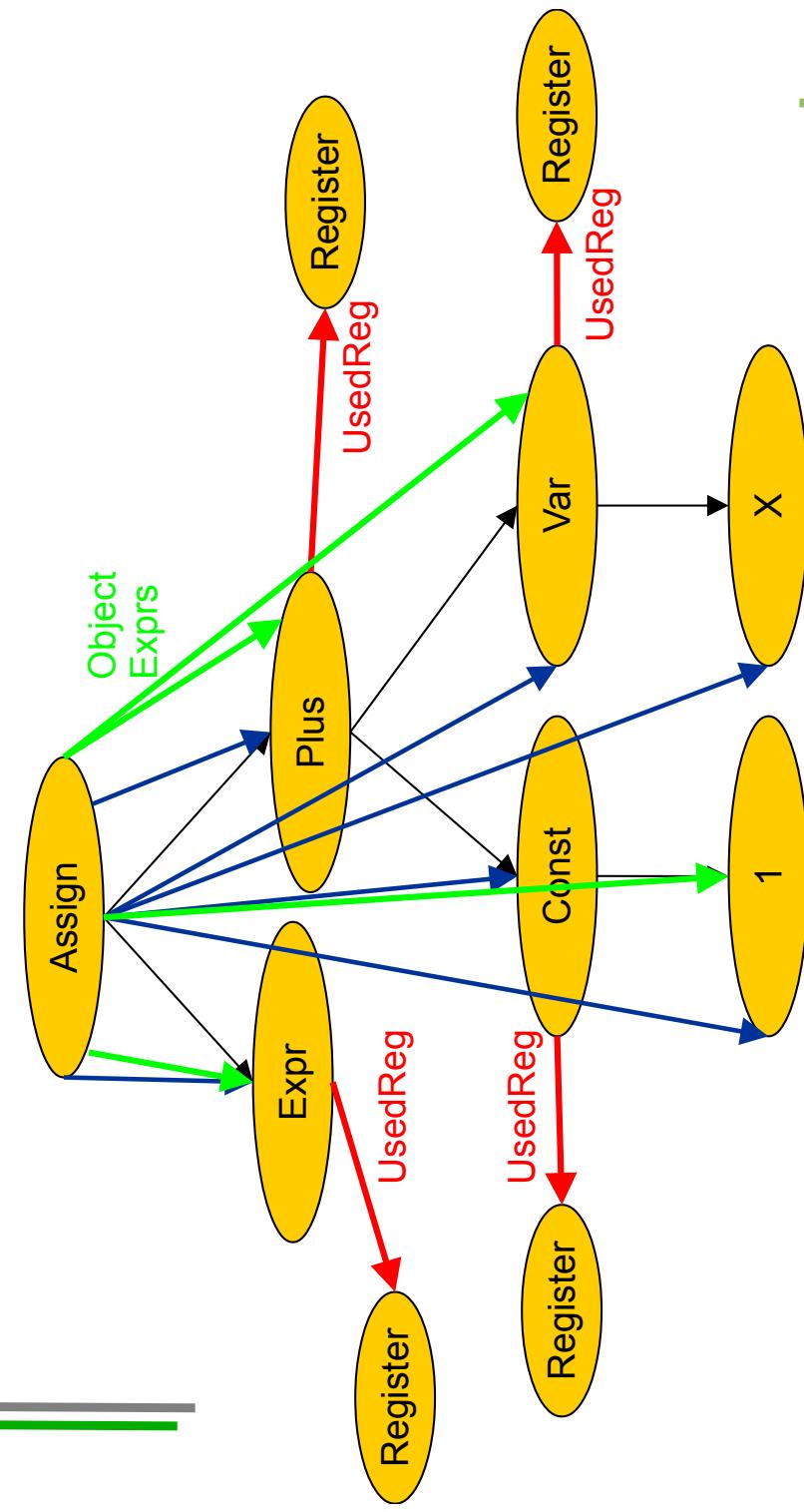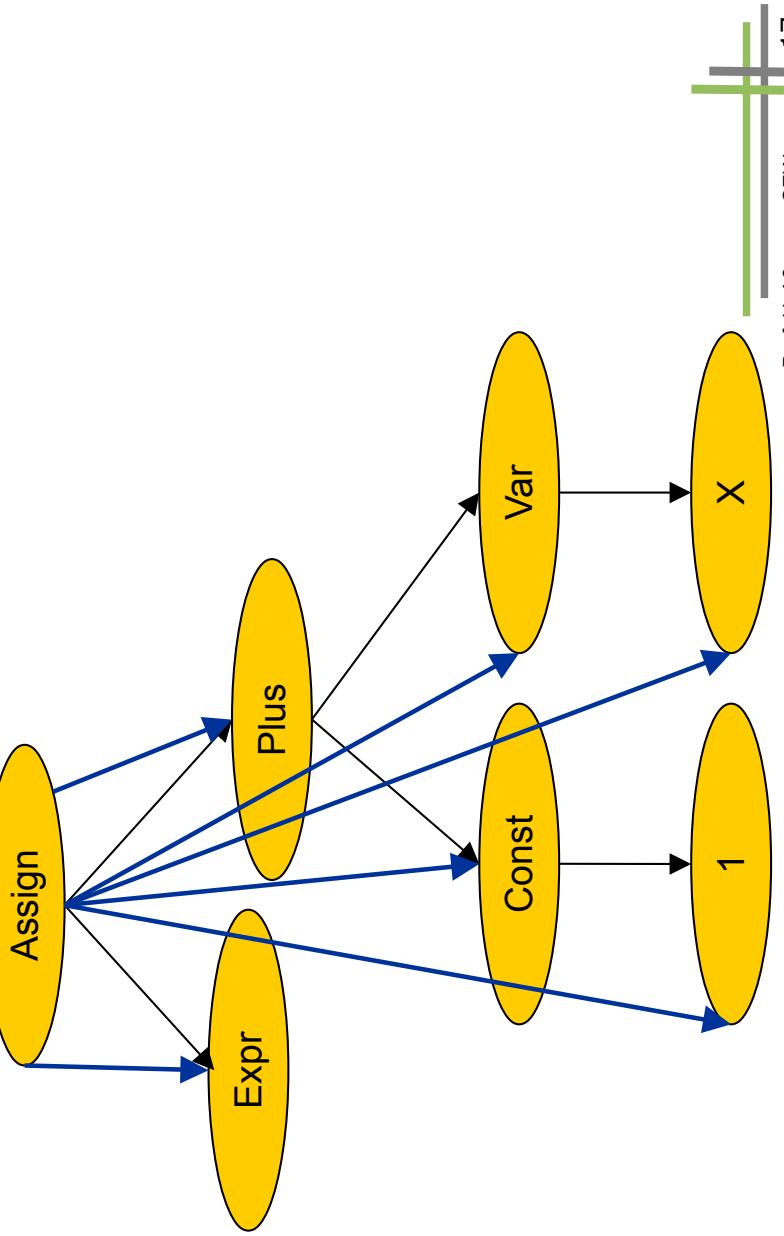
if ExprsOfStmt(Stmt,Expr), HasResult(Expr)

then

ObjectExprs(Stmt,Expr),

RegisterObject := new Register;

UsedReg(Expr,RegisterObject)

..
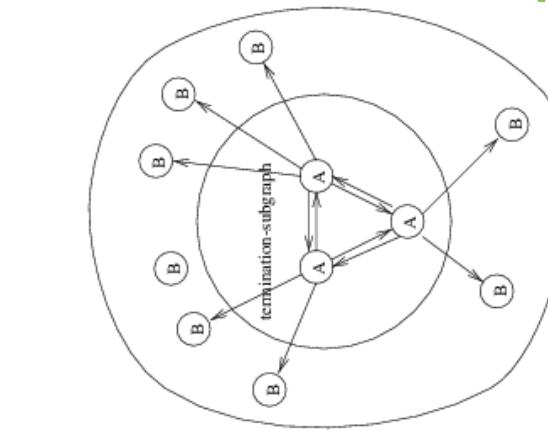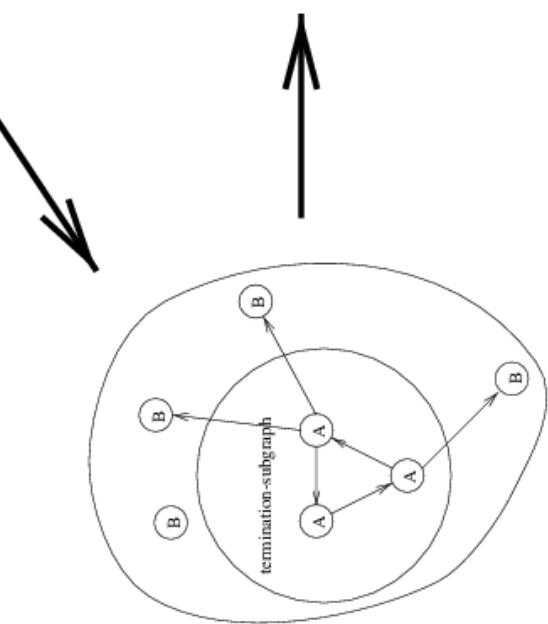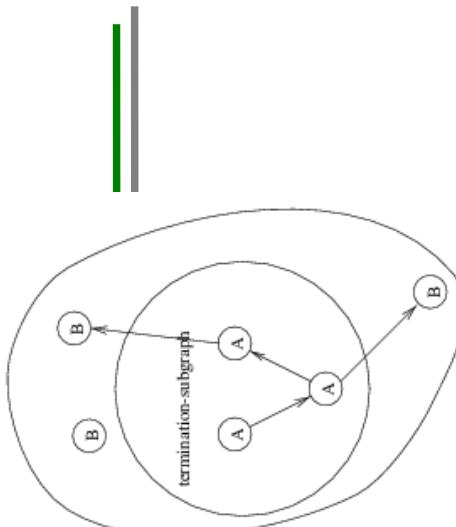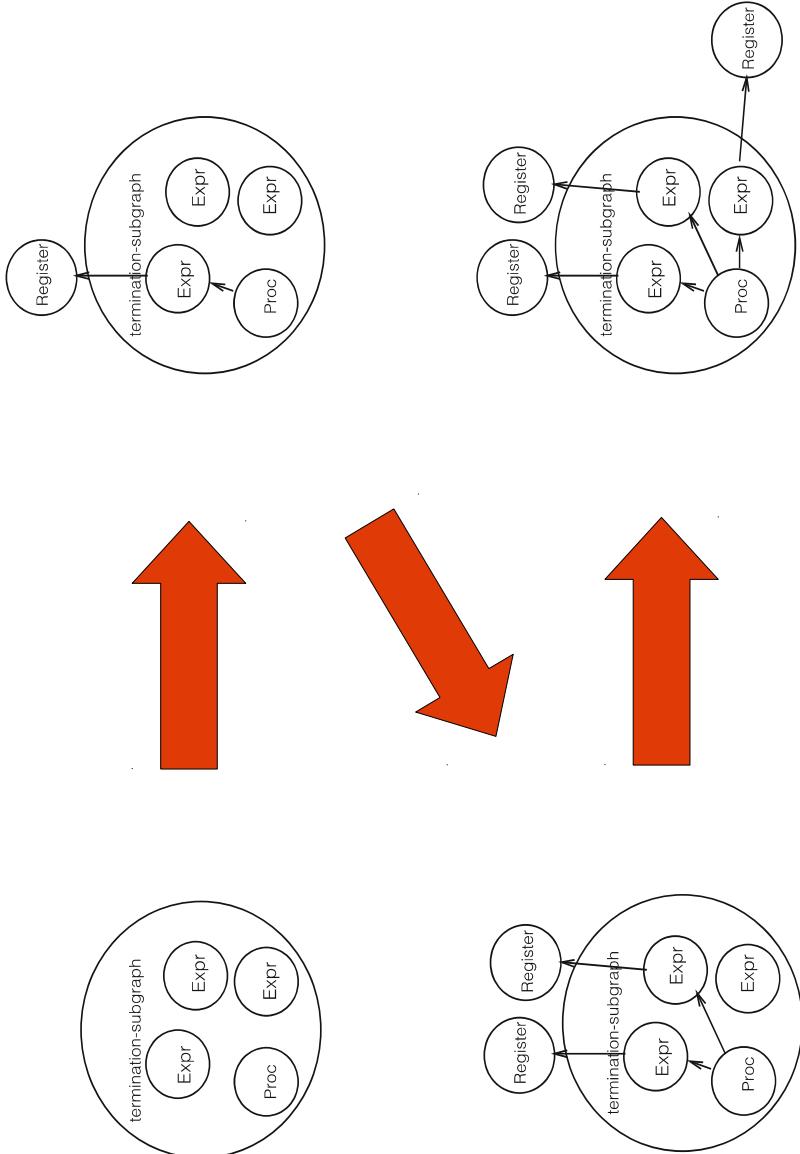
- Features: terminating

ObjectExprs is the termination subgraph

Derivation with the Termination Subgraph

[Aßmann00]

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

Prof. U. Aßmann, SEW

The Termination Subgraph of the Examples

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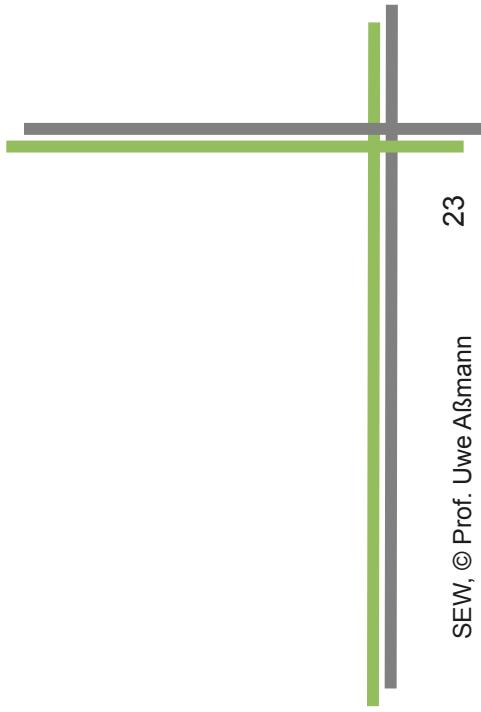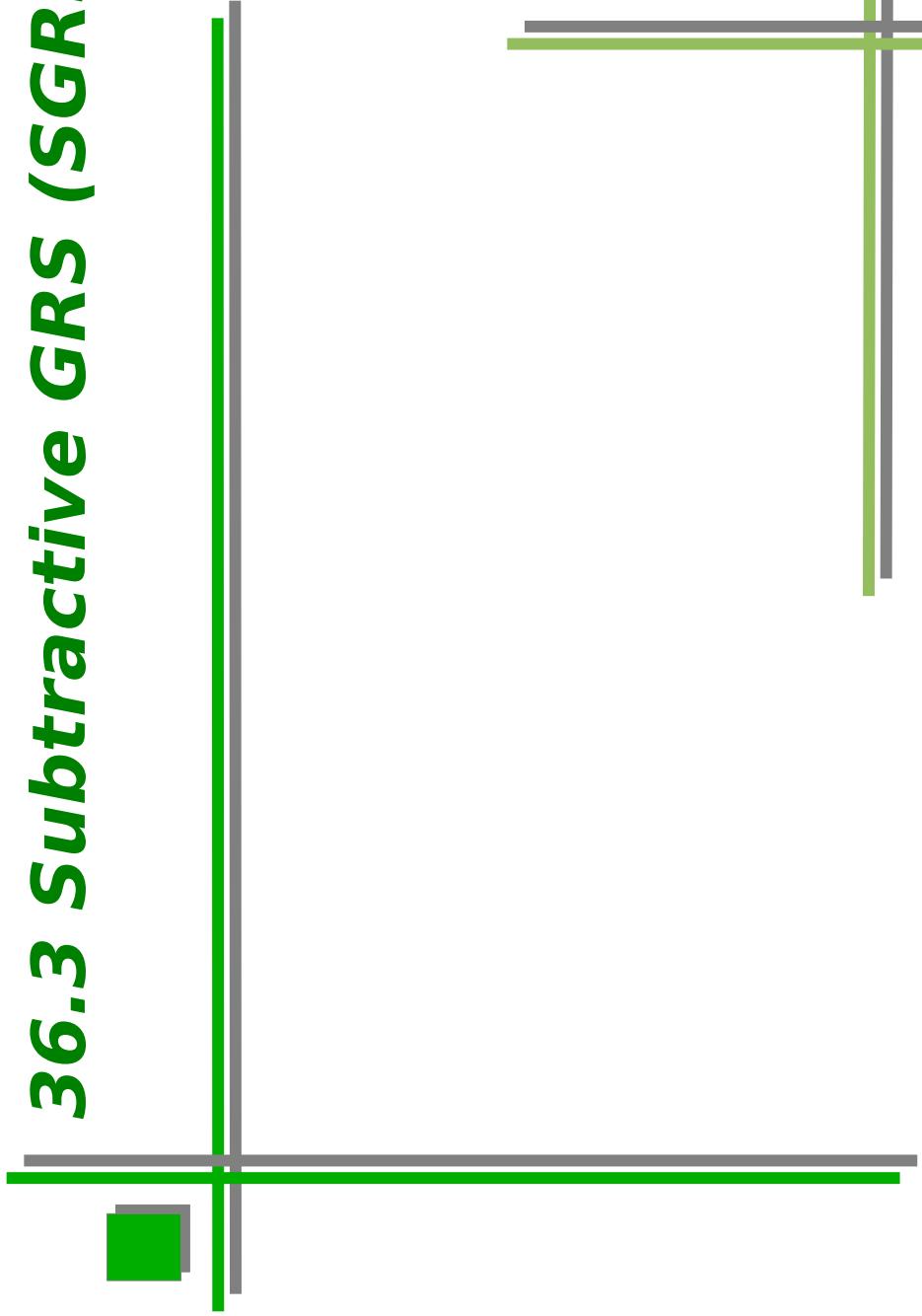
T = ({Stmt,Expr}, {ExprsOfStmt, Descendant})

Allocation of register objects:

T = ({Proc,Expr}, {ObjectExprs})

Prof. U. Aßmann, SEW
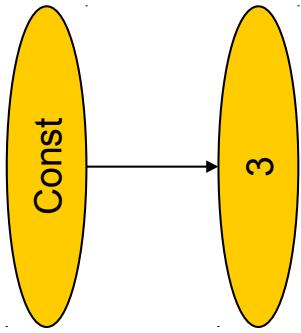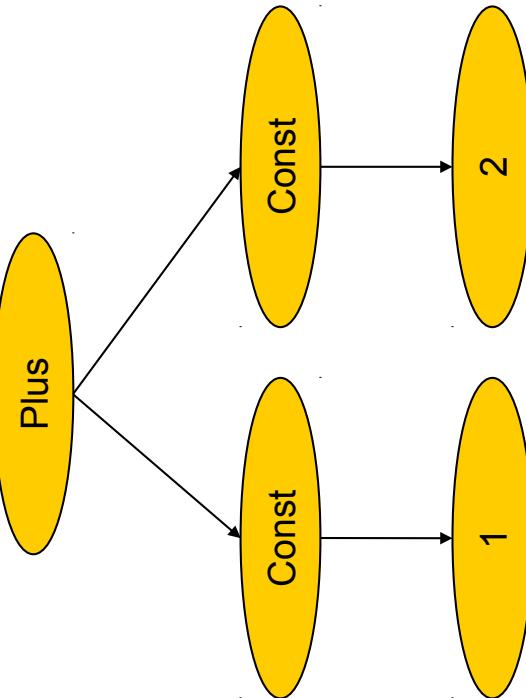
36.3 Subtractive GRS (SGRS)
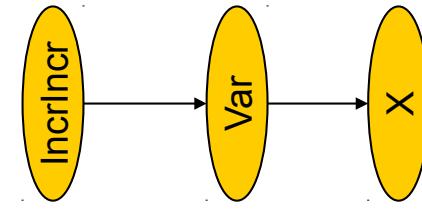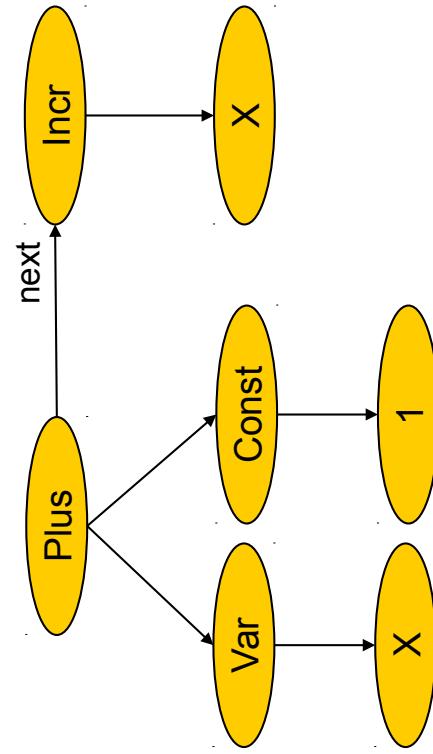
Subtractive Termination

▲ 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)**
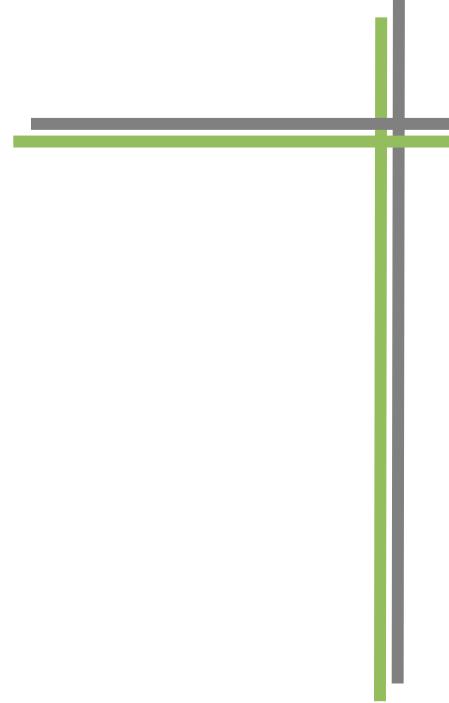
Constant Folding as Subtractive GRS

Peephole Optimization as Subtractive XGRS

36.4 Exhaustive GRS (XGRS)

## The Nature of Exhaustive Graph Rewriting (XGRS)

AGRS, SGRS make up **XGRS** (**eXhaustive Graph Rewrite Systems**)
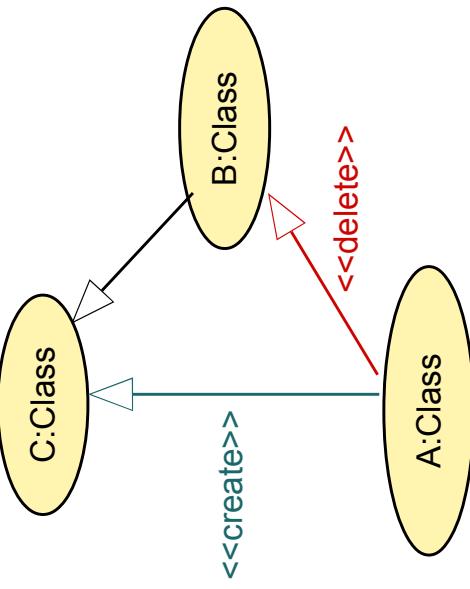
All redex parts in the termination-subgraph of the host graph are reduced step by step.

▲ 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.

# All Together Now: Flattening the Inheritance Hierarchy
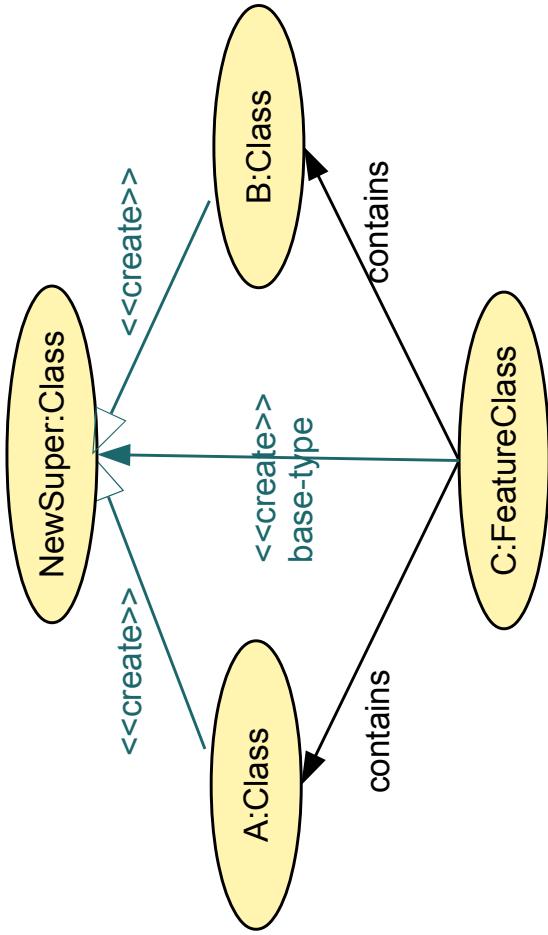
▶ This rule terminates, due to path contraction

```
        C:Class
       ◁      ◁
      /        \
  <<create>>    B:Class
    /            △
 A:Class          \
    \              <<delete>>
     \            /
      ----------
```

A:Class —<<create>>→ C:Class
B:Class —▷ C:Class
A:Class —<<delete>>→ B:Class
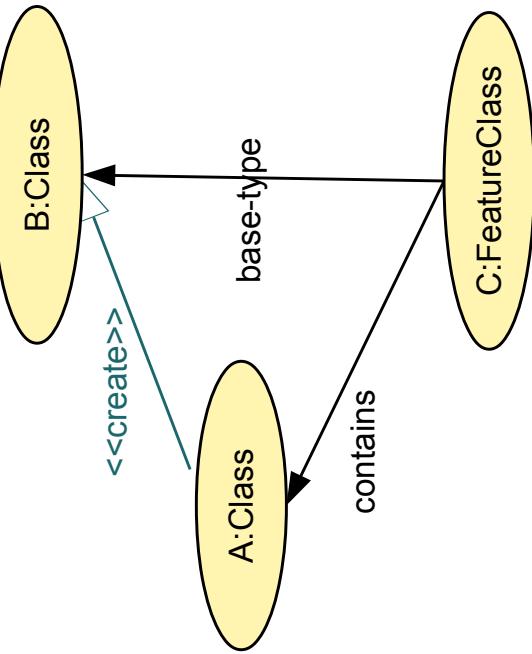
All Together Now: Pull-Up-Method Refactoring

▶ Additive Step 1: Create a new base class for common features; mark this as "base-type"

NewSuper.Name := "<A.name>_<B.name>_Base"

```
    A:Class          B:Class
       \            /
    <<create>>   <<create>>
         \        /
       NewSuper:Class
            ▲
        <<create>>
        base-type
            |
       C:FeatureClass
```

A:Class —contains→ C:FeatureClass
B:Class —contains→ C:FeatureClass

[Christoph04]
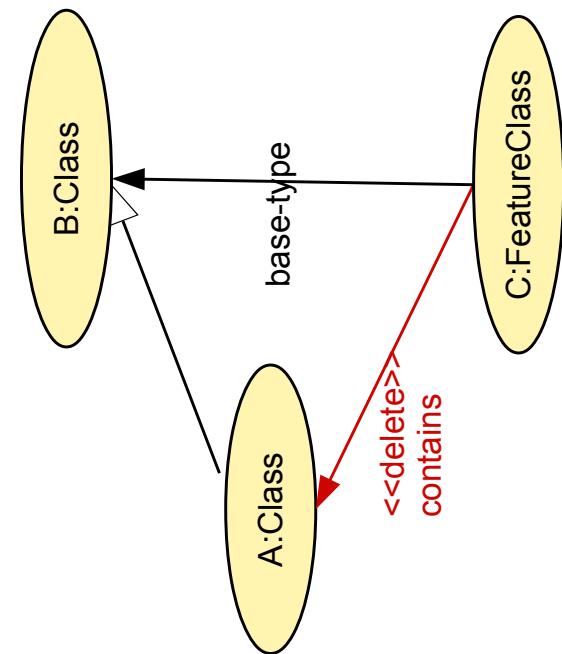
▲ Edge-Additive Step 2: alternate case: a class A has features that should be moved up anyway

B:Class

base-type

<<create>>

A:Class

C:FeatureClass

contains

▲ Subtractive Step 3: do the real "pull-up" into the superclass

{ forall f in C: move f to B }

B:Class

base-type

A:Class

<<delete>>
contains

C:FeatureClass

The End

- ▲ Many model and program transformations can be specified by XGRS

- ▲ Termination criteria build on a *termination subgraph* that is completed or deleted during the transformation

Prof. U. Aßmann, SEW