

41. Deep Graph Model Analysis and Macromodels: Model and Program Analysis with (Recursive) Graph Reachability

How Context-Sensitive Constraints can be Checked in a Model

Prof. Dr. Uwe Aßmann

Softwaretechnologie

Technische Universität Dresden

Version 21-1.1, 29.01.22

- 1) Graph Reachability as Deep Analysis
- 1) EARS
 - 1) Regular graph reachability and Slicing
- 2) Graph slicing
- 3) Value-flow analysis
 - 1) Context-free graph reachability
- 4) More on the Graph-Logic Isomorphism
 - 1) Implementation in Tools
- 5) Model Mappings in Megamodels

Literature

- ▶ GrGen web site <http://www.info.uni-karlsruhe.de/software/grgen/>
- ▶ GrGen User Manual
<http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf>
- ▶ [Aßmann00] 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>
- ▶ Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
 - <http://www.springerlink.com/content/5742246115107431/>
- ▶ Thomas Reps. Program analysis via graph reachability. Information and Software Technology, 40(11-12):701-726, November 1998. Special issue on program slicing.
- ▶ Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352-357, July 1984.
- ▶ Frank Tip. A survey of program slicing techniques. Journal of Programming Languages, 3:121-189, 1995.

Literature on the Graph-Logic-Isomorphism

- ▶ B. Courcelle. Graphs as relational structures: An algebraic and logical approach. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, 4th International Workshop On Graph Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 238-252. Springer, March 1990.
- ▶ B. Courcelle. The logical expression of graph properties (abstract). In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, 4th International Workshop On Graph Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 38-40. Springer, March 1990.
- ▶ B. Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, Handbook of Theoretical Computer Science, pages 193- 242, Amsterdam, 1990. Elsevier Science Publishers.

Other References

- ▶ Uwe Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In Graph Grammar Handbook, Vol. II. Chapman-Hall, 1999.
- ▶ K. Lano. Catalogue of Model Transformations
 - <http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf>

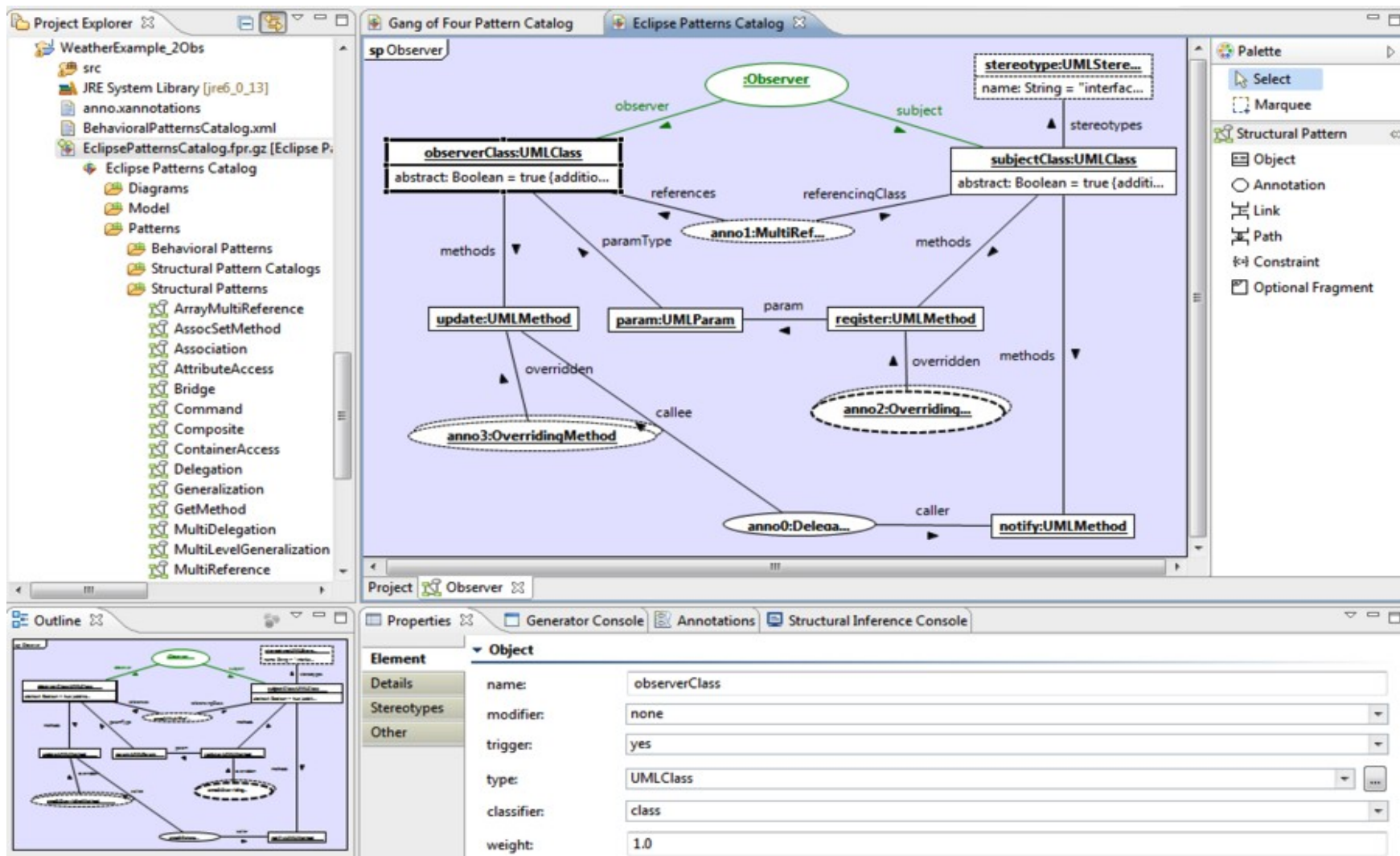
41.1. Introduction to Diagrammatic Storyboard Rule Notation for Graph Rewriting

Originally introduced by Fujaba www.fujaba.de (tool now unsupported)



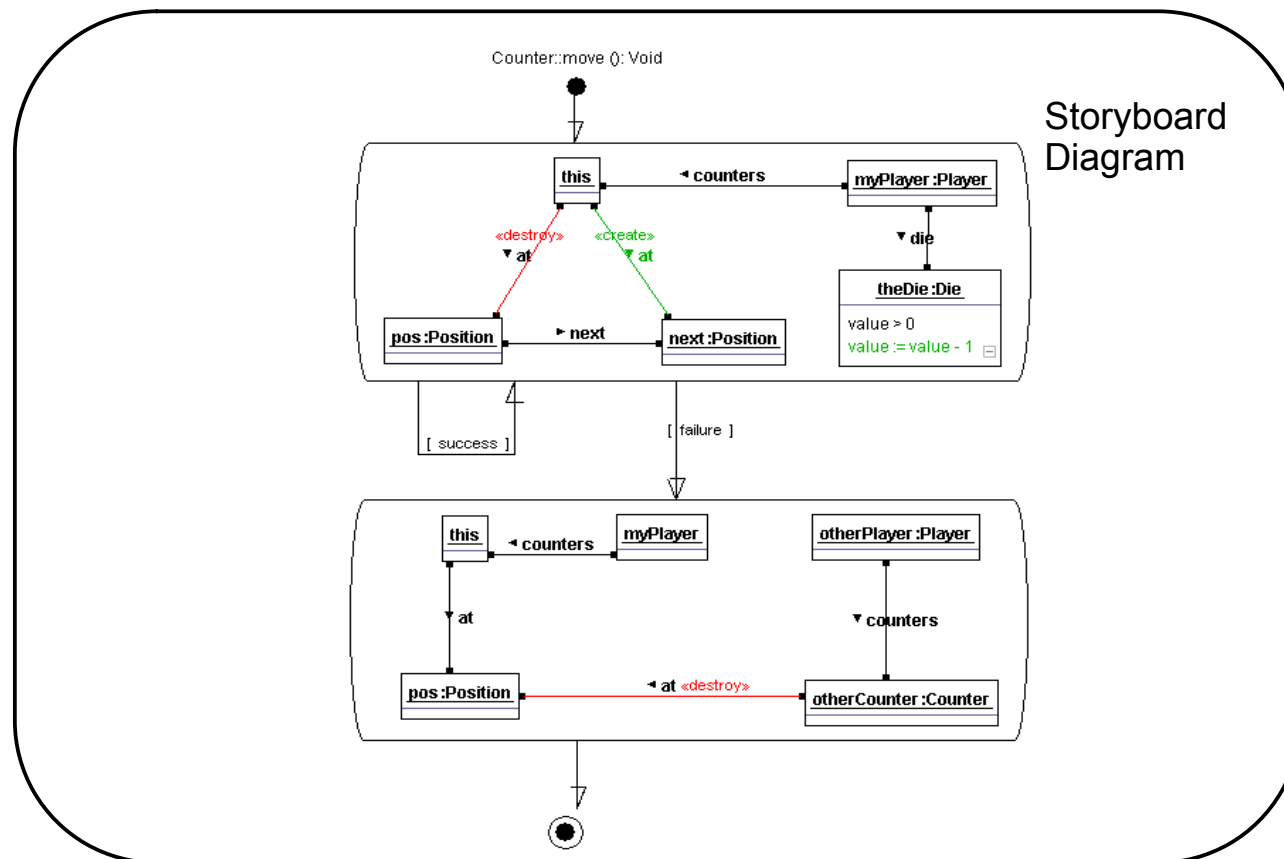
Fujaba

- ▶ Fujaba is a MetaCASE-tool based on GRS with home-grown metalanguage and metamodel
- ▶ Basic technology: graph pattern matching and rewriting



Fujaba Storyboard Diagrams for Adding and Removing Graph Fragments

- ▶ Storyboards are activity diagrams in which activities are GRS (graph notation with colors)
- ▶ **Green** color: adding model fragments; **Red** color: deleting them
- ▶ Pool starts at node this and reaches into the object net
- ▶ GRS can be embedded into Petri Nets, DFG and other BSL



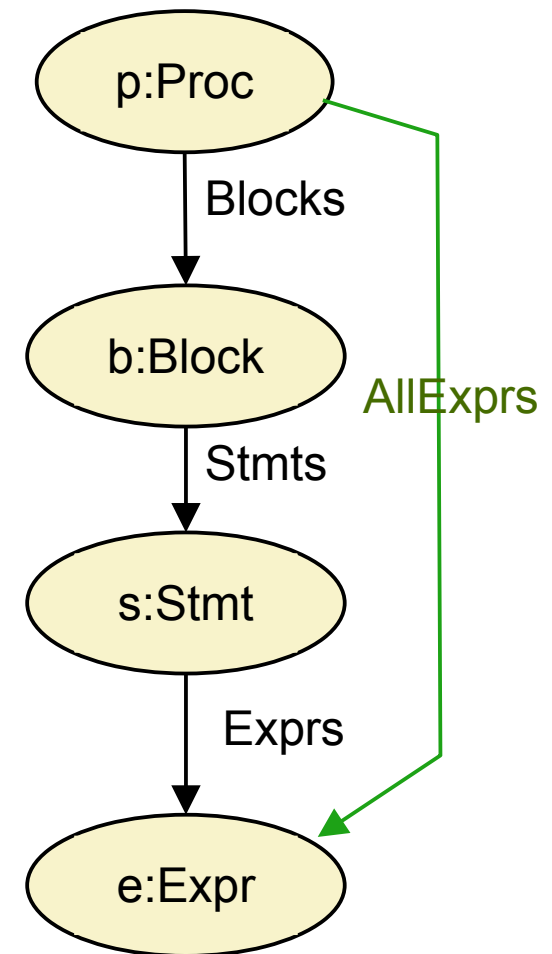
41.1 Using EARS (Binary Edge Addition) for Deep Analysis of Models and Mappings of Models and Code

- and as the Bridge to Graph Rewriting

- Graph reachability engines are analysis tools answering questions about the deeper structure of models and programs
- EARS can be employed for regular graph reachability, context-free graph reachability, slicing, data-flow analysis
- And traceability for inter-model relationships

EARS for Model Mapping

- ▶ **Edge addition rewrite systems (EARS)** compute direct binary relations for remotely reachable parts of a graph and a model
 - They **abbreviate long paths** in models
- ▶ EARS can be used for reachability of elements in models and model mapping:
 - Transitive closure
 - Regular path reachability
 - Context-free path reachability
- ▶ EARS form the bridge to graph rewriting and graph-rewriting based model transformations
- ▶ They correspond to binary Datalog



Model Analysis with Graph Reachability

- ▶ Use the **graph-logic-isomorphism**: Represent everything in a program or a model as directed graphs
 - Program code (control flow, statements, procedures, classes)
 - Model elements (states, transitions, ...)
 - Analysis information (abstract domains, flow info ...)
 - Directed graphs with node and edge types, node attributes, one-edge condition (no multi-graphs)
- ▶ **Use edge decomposition as textual notation**
- ▶ Use edge addition rewrite systems (EARS), Datalog and other graph reachability specification languages to
 - Query the graphs (on values and patterns)
 - Analyze the graphs (on reachability of nodes)
 - Map the graphs to each other (model mapping)
- ▶ Later: Use graph rewrite systems (GRS) to construct and augment the graphs, transform the graphs

Specification Process with Graph Rewrite Systems and EARS

1) Specification of the data model (graph schema) with a graph-like DDL (ERD, MOF, GXL, UML or similar):

- **Schema of the program representation:** program code as objects and basic relationships. This data, i.e., the start graph, is provided as result of the parser
- **Schema of analysis information** (the inferred predicates over the program objects) as objects or relationships

2) Flat model and program analysis (preparing the abstract interpretation)

- Querying graphs, enlarging graphs, static slicing, Reachability
- Equivalence classing
- Materializing implicit knowledge to explicit knowledge

3) Deep model and program analysis

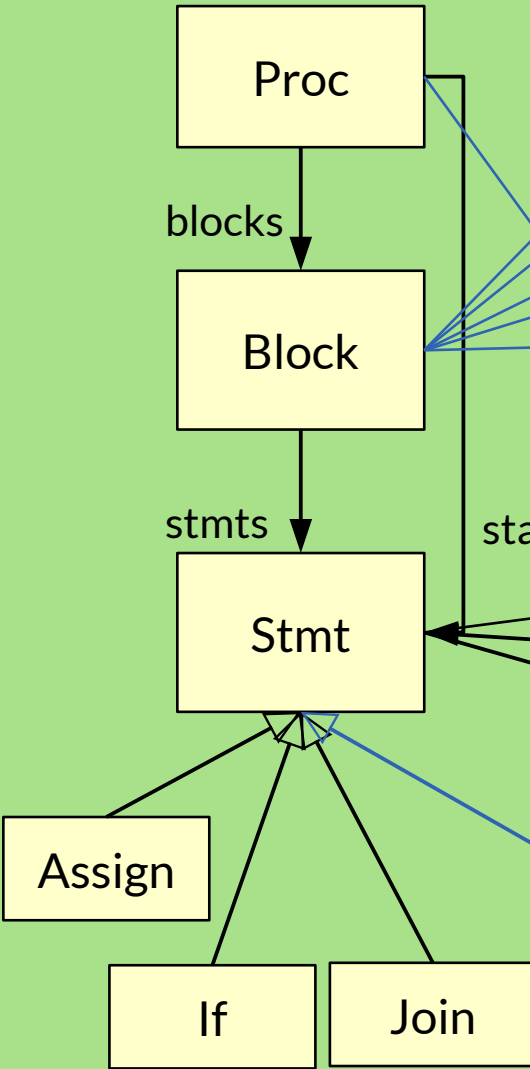
- Inter-model reachability (traceability), materializing model mappings
- **Abstract Interpretation** (program analysis as interpretation)
- Specifying the transfer functions of an abstract interpretation of the program with graph rewrite rules on the analysis information

4) Model and Program transformation Transforming the program representation

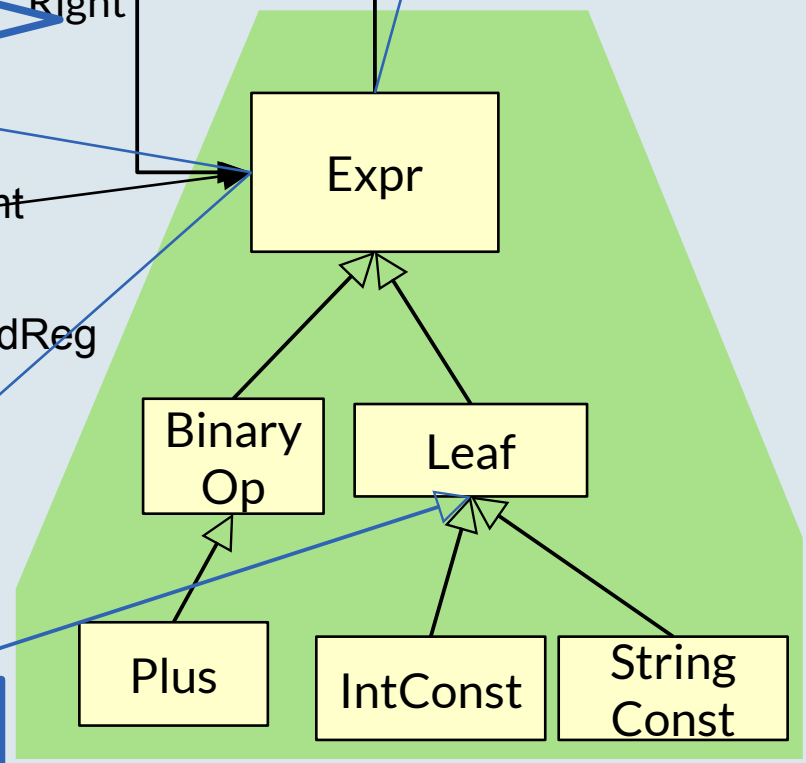
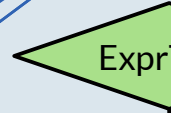
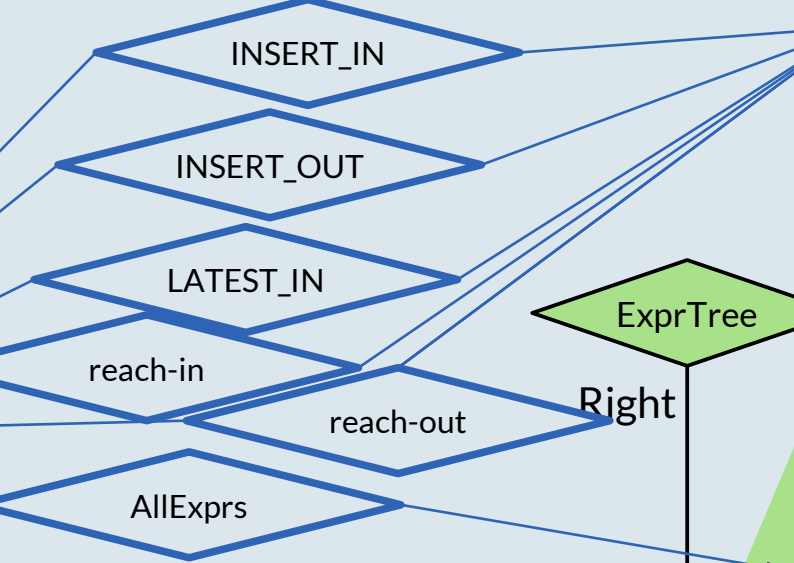
- Optimization such as peephole optimization or constant folding (context-free)
- Code motion (Context-sensitive)

Q14: A Simple Program (Code) Model (Schema) in MOF

Program representation:
ProgramNode (green)



Analysis information: InfoNode (blue)



statements

ExprsOfStmt

successors

predecessors

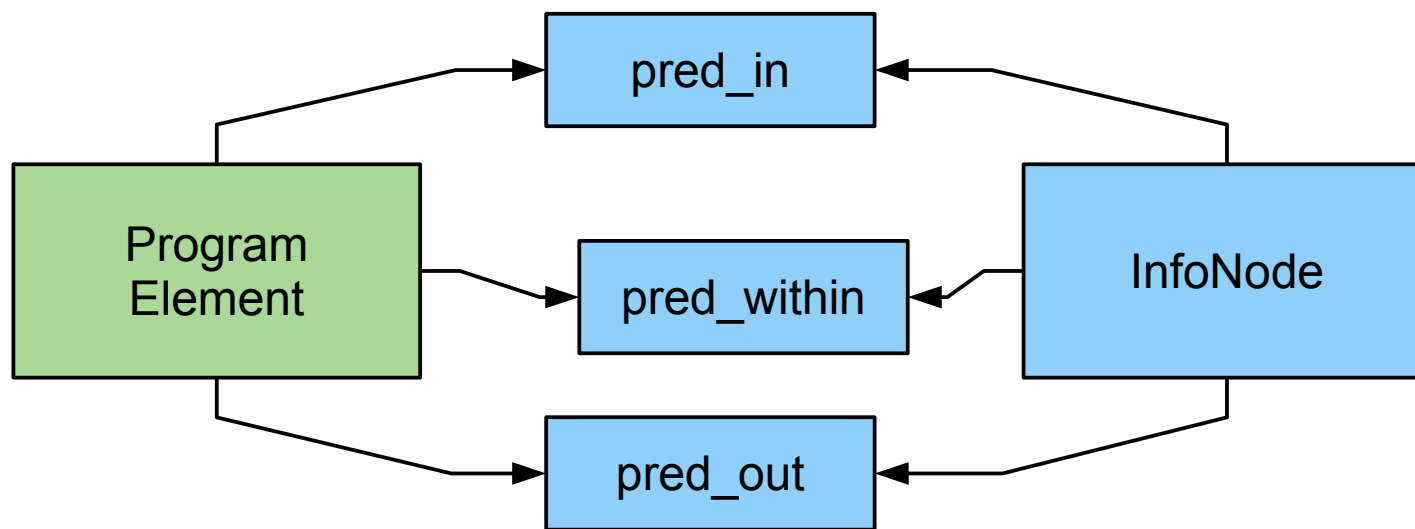
ControlFlowGraph

AsgdReg

UseReg

Deep Analysis and Abstract Interpretation with Graph Rewriting

- ▶ A **graph-rewriting based abstract interpreter** stores, for every program element of the program graph (Expr, Stmt, Block, Proc, Class < ProgramElement) three “truths” (values) for every node in the analysis information (InfoNode):
 - `p:ProgramElement -:predicate_in-> i:InfoNode`
 - `// predicate_in(p,i)`
 - `p:ProgramElement -:predicate_within-> i:InfoNode`
 - `// predicate_within(p,i)`
 - `p:ProgramElement -:predicate_out-> i:InfoNode`
 - `// predicate_out(p,i)`
- ▶ Values of program elements are encoded as an edge between program elements and InfoNodes



41.2. Reachability of Model Elements and Models for Model Analysis and Mapping

- ▶ With model mapping languages, such as edge addition rewrite systems or TGreQL



41.2.1. Simple Reachability of Model Elements and Models: Path Abbreviations in Graph Analysis

- ▶ With model mapping languages, such as edge addition rewrite systems or TGreQL

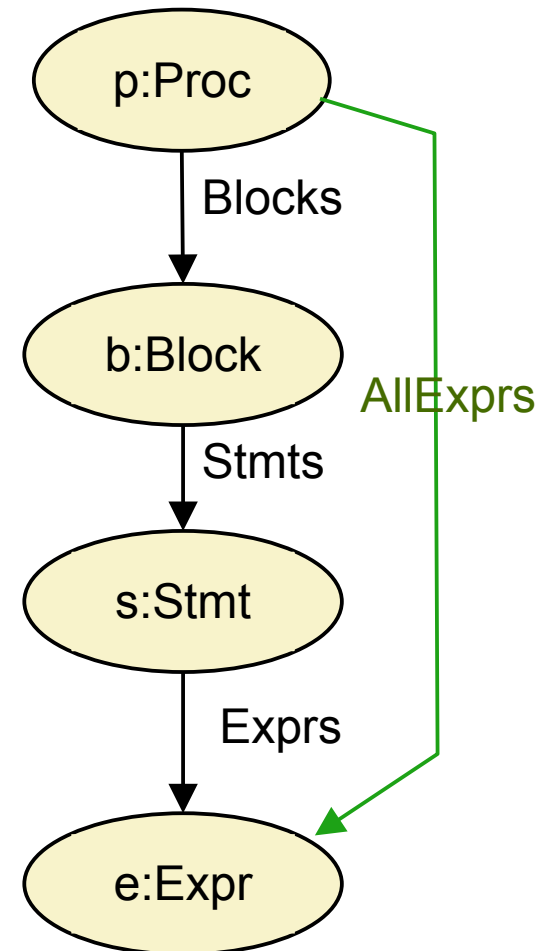


Path Abbreviations for Simple Reachability

- ▶ Path abbreviations shorten paths in the manipulated graph.
- ▶ They may collect nodes into the neighborhood of other nodes.
- ▶ Ex.: Collection of Expressions for a procedure: edge addition

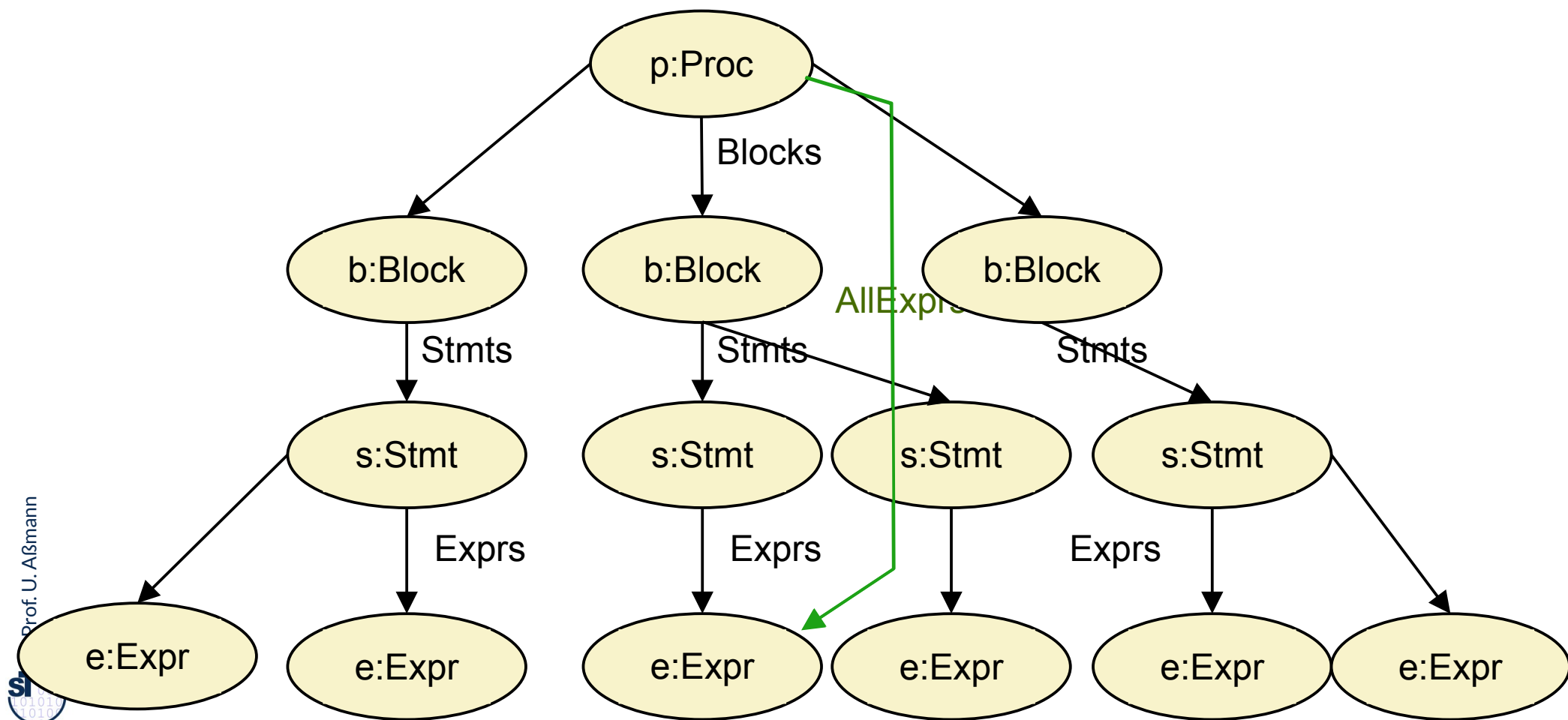
```
-- GrGen notation:  
rule collectAllExpr(p:Proc) {  
  p -:Blocks-> b:Block;  
  b -:Stmts-> s:Stmt;  
  s -:Exprs-> e:Expr;  
  modify {  
    p -:AllExprs-> e;  
  }  
}
```

```
-- F-Datalog notation (edge decomposition):  
AllExprs(Proc, Expr) :-  
  Blocks(Proc, Block),  
  Stmts(Block, Stmt),  
  Exprs(Stmt, Expr).  
-- if-then rules:  
if Blocks(Proc, Block),  
  Stmts(Block, Stmt),  
  Exprs(Stmt, Expr)  
then  
  AllExprs(Proc, Expr);  
- regular expression notation (TGreQL):  
AllExprs := Proc Blocks.Stmts.Expr Expr
```



Forward Slicing from a Point in the Program Graph (Single-Source Multiple-Target (SSMT) Problems)

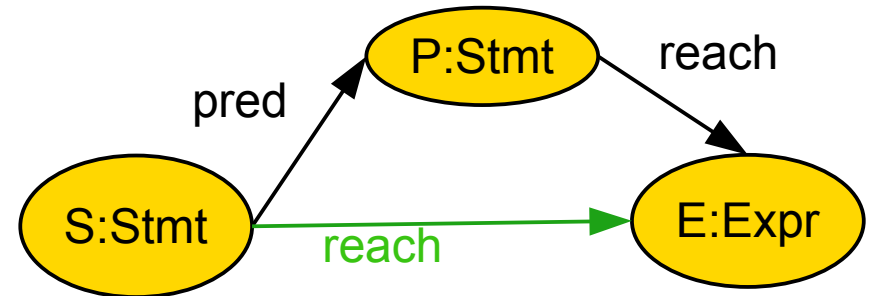
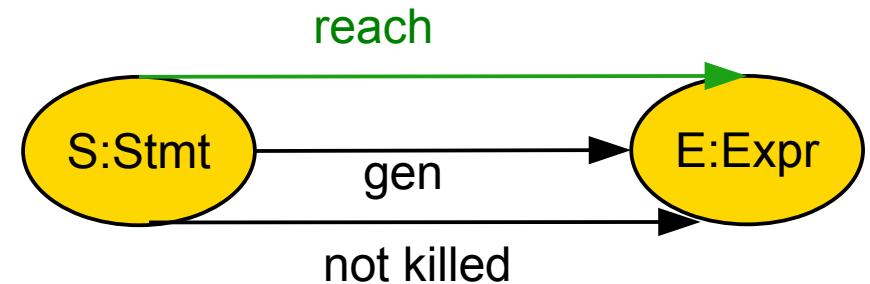
- ▶ A **forward slice** (SSMT-region) has one source, many targets and all intermediate nodes
- ▶ The slice border is the border of the region



Linear Recursion for Remote Reachability



- ▶ Reachability most often can be reduced to transitive closure of one or several relations.
- ▶ **Query: "Does an Stmt S reach a expression E?"**
- ▶ TC combines path abbreviation with recursion
 - F-Datalog, GrGen: Left or right recursion
 - Kleene * in TGreQL
 - Thick arrow in Fujaba



```
// TGreQL
reach[gen]*(S:Stmt,E:Expr)

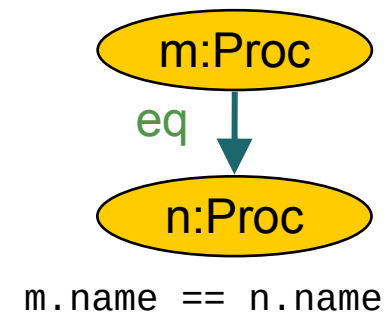
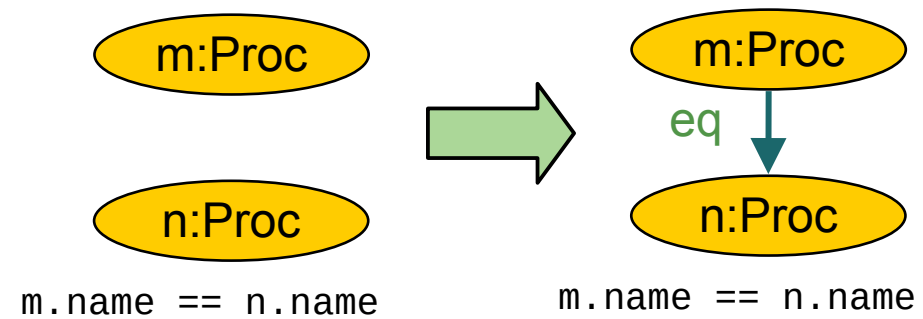
// GrGen can use inheritance on
// nodes and edges
rule reachability (s:Node) {
  s -:BasicEdge-> p:Node;
  p -:RecursiveEdge-> e:Node;
  modify {
    s -:RecursiveEdge-> e:Node
  }
}
```

```
// F-Datalog
reach(S:Stmt,E:Expr) :- gen(S:Stmt,E:Expr), not killed(S:Stmt,E:Expr).
reach(S:Stmt,E:Expr) :- pred(S:Stmt,P), reach(P,E:Expr).
```

Ex.: Relating Nodes into Equivalence Classes

- ▶ Ex.: Computing equivalent nodes
- ▶ Context-sensitive problem, because m is not in the context of n

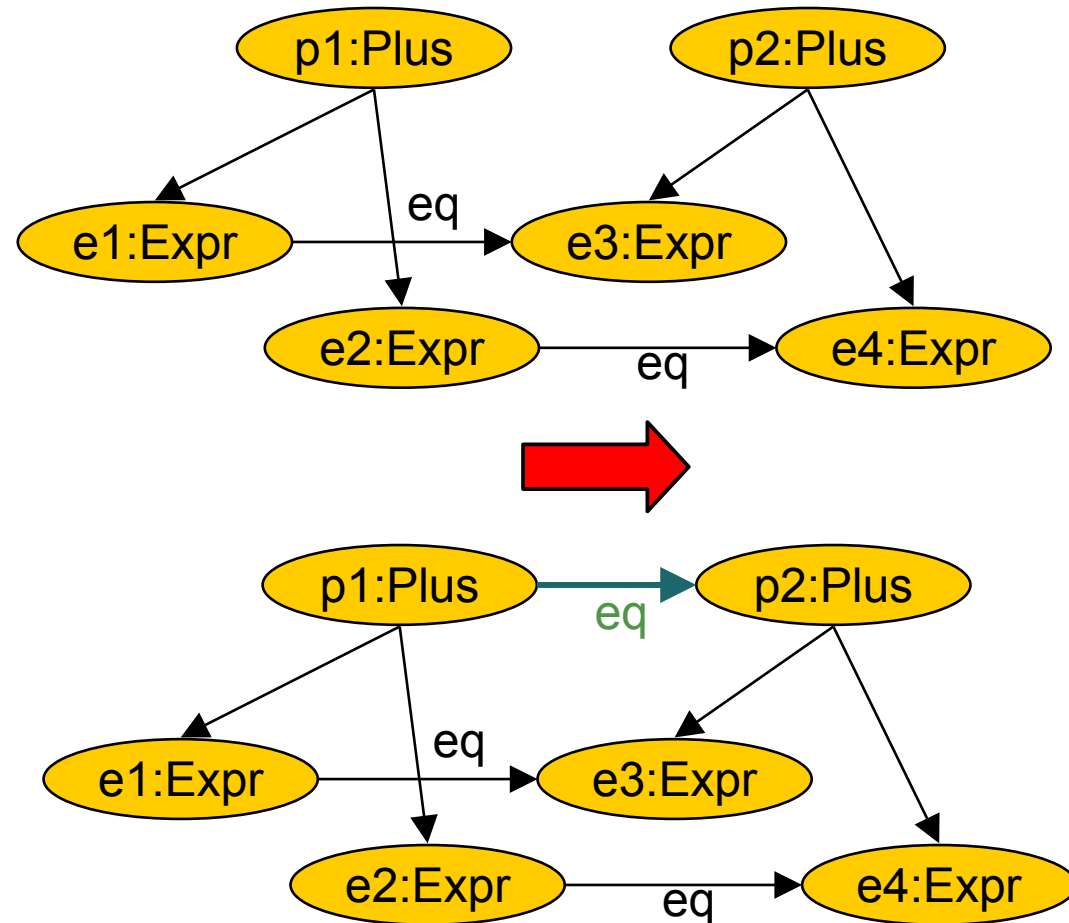
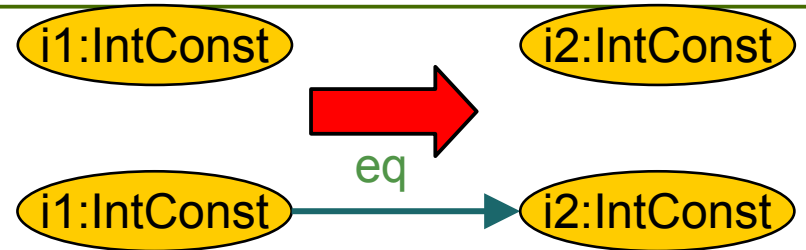
```
F-Datalog baserule:  
eq(m:Proc, n:Proc) :-  
  m.name == n.name.  
-- If-then:  
If (m:Proc, n:Proc) and m.name == n.name)  
  eq(m, n)  
}  
- TgreQL regular expression:  
m:Proc eq n:Proc if  
m.name == n.name
```



```
// GrGen  
rule buildGraph(m:Node, n:Node) {  
  m.Name == n.Name;  
  modify { m:-eq-> n }  
}
```

Ex. Relating Nodes into Equivalence Classes (Here: Value Numbering, Synt. Expression Equivalence)

- ▶ Ex.: Computing structurally equivalent expressions with bi-recursive reachability
- ▶ Question: "Which expression trees have the same structure?"



```
--- F-Datalog baserule:  
eq(i1: IntConst, i2: IntConst) :-  
  i1 ~= IntConst(Value),  
  i2 ~= IntConst(Value).  
--- recursive_rule:  
eq(p1: Plus, p2: Plus) :-  
  p1 ~= Plus(Type),  
  p2 ~= Plus(Type),  
  Left(p1, e1),  
  Right(p1, e2),  
  Left(p2, e3),  
  Right(p2, e4),  
  eq(e1, e3),  
  eq(e2, e4).
```



41.3. Deep Model Analysis (Value-Flow Analysis, Data-Flow Analysis) as General (Recursive) Graph Reachability over Values

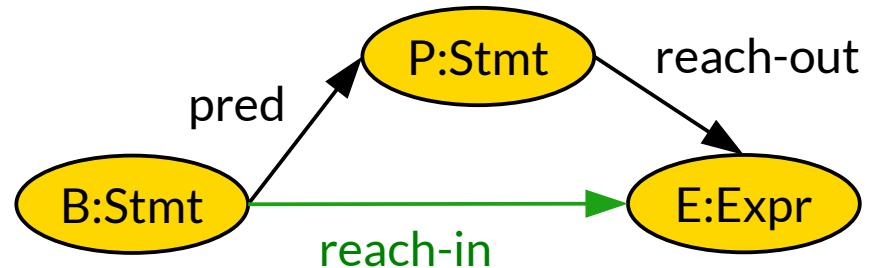
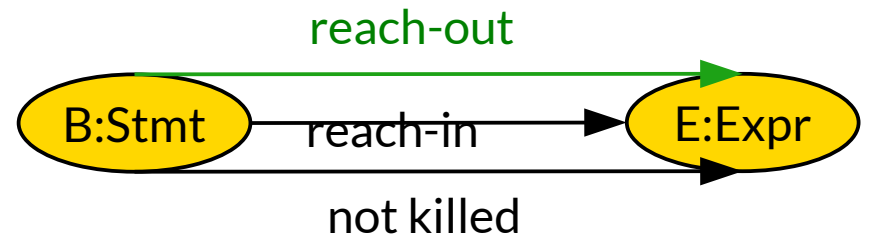
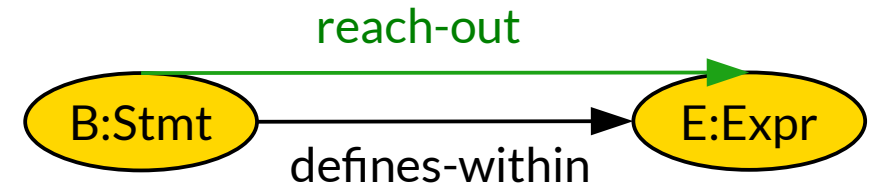
- with edge addition rewrite systems and F-Datalog

Data-flow Analysis for Reachability and Traceability

- ▶ **Value-flow analysis (data-flow analysis)** is a specific form of deep model analysis asking **reachability questions**, i.e., computing the *flow of data (value flow)* through the model or program, from variable assignments to variable uses
 - Result: the **value-flow graph (data-flow graph)**
 - If the value flow analysis is done along the control-flow graph, it is called an **abstract interpretation** of a program
 - EARS can do an abstract interpretation of a program, if they are rewriting on the control-flow graph. Then, their rules implement transfer functions of an abstract interpreter
- ▶ Examples of reachability problems:
 - **AllSuperClasses**: find out for a class transitively all superclasses
 - **AllEnclosingScopes**: find out for a scope all enclosing scopes
 - **AllEnclosingWholes**: find out, for a part, its wholes into which it is included
 - **Reaching Definitions Analysis**: Which Assignments (Definitions) of a variable can reach which statement?
 - **Live Variable Analysis**: At which statement is a variable live, will further be used?
 - **Busy Expression Analysis**: Which expression will be used on all outgoing paths?

Reaching Definition Analysis By Abstract Interpretation with EARS (Reachable Statements from Expression Definition)

- ▶ **Query:** "Which definitions of expressions reach which statement?"
 - Assignments of a variable, temporary, or register
 - Usually computed for all positions *before* and *after* a statement
- ▶ Graph rewrite rules implement an abstract interpreter
 - On instructions or on blocks of instructions
 - Flow information is expressed with edges of relations "reach-*"
- ▶ Recursive system (via edge reach-in)
 - $(B \text{ reach-out } E) := (E \text{ reaches end of block } B)$
- ▶ GrGen can express this via its generic reachability rules



```
reach-out(B,E) :- defines-within(B,E).
reach-out(B,E) :- reach-in(B,E), not killed(B,E).
reach-in(B,E) :- pred(B,P), reach-out(P,E).
```

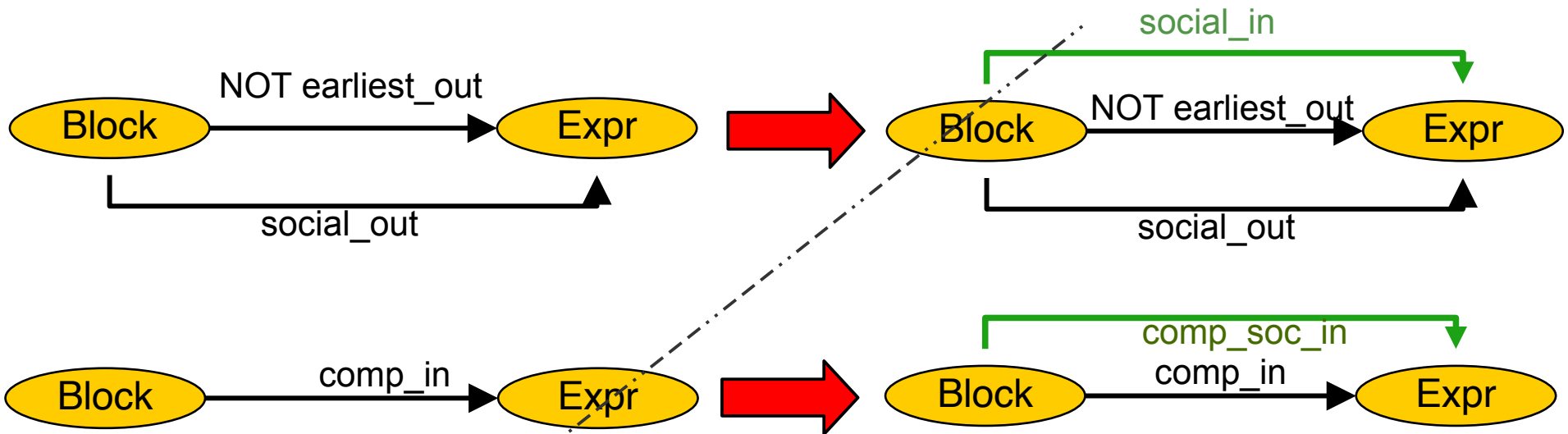
Code Motion Analysis

- ▶ **Code motion** is an essential transformation to speed up the generated code. However, it is a complex transformation:
 - Discovering loop-invariant expressions by data-flow analysis
 - Moving loop-invariant expressions out of loops upward
 - Code motion needs complex data-flow analysis
- ▶ **Busy Code Motion (BCM)** moves expressions as upward (early) as possible
- ▶ **Lazy Code Motion (LCM)**
 - Moving expressions out of loops to the front of the loop, upward, but carefully:
 - Moving expressions to an optimal place so that register lifetimes are shorter and not too long (optimally early)
 - LCM analysis computes this optimal early place of an expression [Knoop/Steffen]
 - Analyze an optimally early place for the placement of an expression
 - About 6 equation systems similar to reaching-definitions
 - Every equation system is an EARS [Aßmann00]

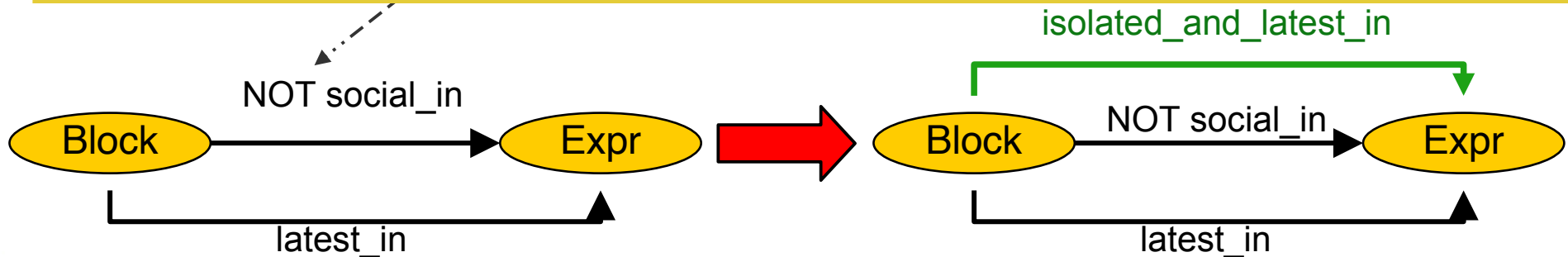
Excerpt from LCM Analysis with Overlaps

- ▶ Compute an optimally early block for an expression (out of a loop)

Query: "Which expression is not isolated (social) at the beginning of a block?"



Query: "Which expression is not isolated (social) at the beginning of a block?"



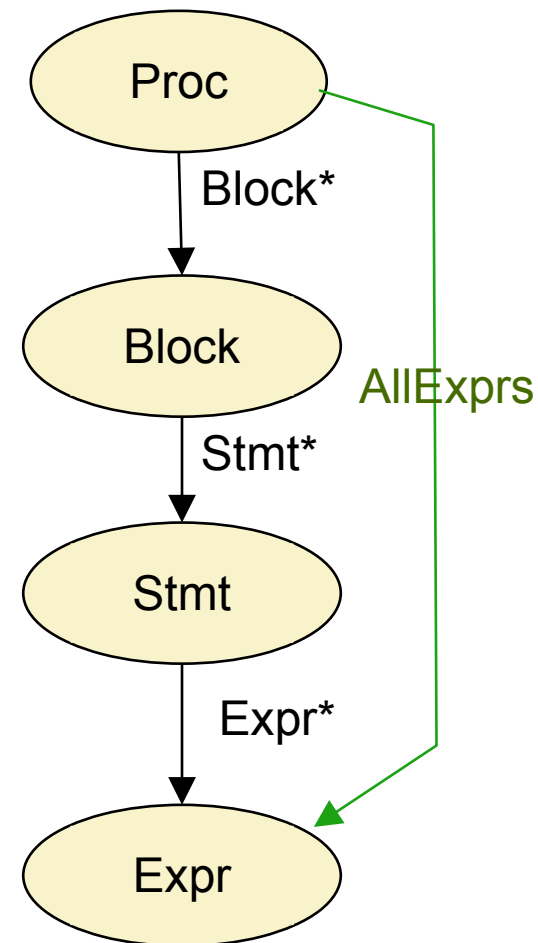
41.3.2 Regular Graph Reachability and Slicing



Regular Graph Reachability

- ▶ If the query can be expressed as a regular expression, the query is a **regular graph reachability problem**
- ▶ Kleene star is used as transitive closure operator
- ▶ TqreQL and Fujaba are languages offering Kleene *

```
-- F-Datalog notation:  
AllExprs(Proc, Expr) :-  
    Block*(Proc, Block),  
    Stmt*(Block, Stmt),  
    Expr*(Stmt, Expr).  
  
-- GrGen if-then rules:  
if Proc -:Block*-> Block,  
   Block -:Stmt*-> Stmt,  
   Stmt -:Expr*-> Expr  
modify {  
    Proc -:AllExprs-> Expr  
}  
  
- regular expression notation (TGreQL):  
AllExprs := Proc Block*.Stmt*.Expr* Expr
```



Static Slicing: Single-Source-Multiple-Target Regular Reachability (Regular Reachable Dependencies)

- ▶ [Weiser] [Tip]
- ▶ A **static slice** is the region of a program or model *dependent* from *one source* node (reachable by a regular reachability query in a dependency graph)
 - A static slice is a *single-source path regular reachability problem (SSPP)* on the dependency graph
 - A static slice introduces path abbreviations from one entity to a region
- ▶ A **forward slice** is a dependent region in *forward* direction of the program
 - The uses of a variable
 - The callees of a call
 - The uses of a type
- ▶ A **backward slice** is a dependent region in *backward* direction of the program
 - The assignments which can influence the value of a variable
 - The callers of a method
 - The type of a variable
- ▶ Slicing can map arbitrary entities in programs and models to other entities, based on a regular graph expression

Reachability within Models and Traceability between Models

- ▶ Data-flow analysis (graph reachability, slicing) can be done
 - Intraprocedurally (within one procedure)
 - Interprocedurally (program-wide)
- ▶ **Traceability** is inter-model slicing and graph reachability
 - inter-model: then it creates **trace relations** between requirements models, design models, and code models
 - Intra-megamodel: trace relations can trace dependencies between all models in a megamodel, e.g., in an MDA
- ▶ A **model mapping** is an inter-model trace(-ability) graph
 - Model mappings are very important for the dependency analysis and traceability in megamodels and the construction of macromodels

41.3.3 Context-Free Graph Reachability

- ▶ If arbitrary recursion patterns are allowed in F-Datalog and EARS queries, we arrive at context-free graph reachability.



Free Recursion

- ▶ Transitive closure and regular graph reachability rely on regular recursion (linear recursion) expressible with the Kleene-* on relations
- ▶ Beyond that, F-Datalog and EARS can describe other recursions
 - Context-free recursions
 - Cross-recursions
- ▶ Then, we speak of **context-free graph reachability**
 - A context-free language describes graph reachability
- ▶ Applications:
 - Complex intraprocedural value flow analyses
 - Interprocedural, whole-program analysis
 - Interprocedural IDFS framework (Reps)
 - Model mappings in a megamodel

41.4 More on the Logic-Graph Isomorphism

- ▶ [Courcelle] discovered that many problems can be expressed in logic (on facts) and in graph rewriting (on graphs)

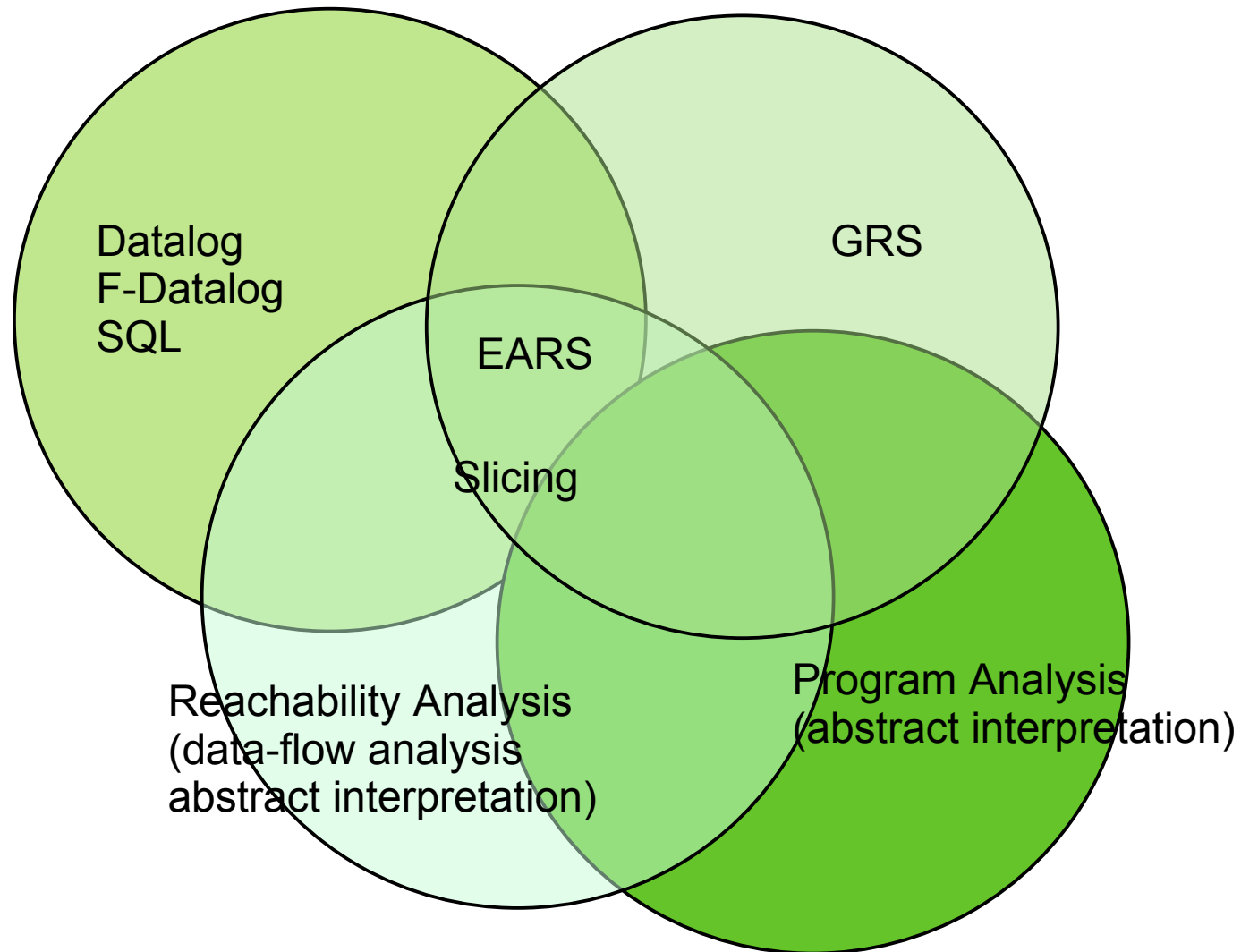


Program and Model Analyses Covered by Graph Reachability

- ▶ Graph Reachability Analysis can do abstract interpretation
 - If it adds analysis information to the program elements and their control-flow graph
 - Slicing is a Single-Source-Multiple-Target reachability analysis
- ▶ Every abstract interpretation where a mapping of the abstract domains to graphs can be found.
 - Monotone and distributive data-flow analysis
Control flow analysis (and callee analysis)
 - Static-single-assignment (SSA) construction
 - Interprocedural IDFS analysis framework (Reps)

The Common Core of Logic, Graph Rewriting and Program Analysis

- ▶ Graph rewriting, DATALOG and data-flow analysis have a common core: EARS
- ▶ Datalog query languages such as .QL or TgreQL can be extended by GRS



Relation DFA/F-DATALOG/GRS

- ▶ Abstract interpretation (Data-flow analysis), F-DATALOG and graph rewrite systems have a common kernel: EARS
 - As F-DATALOG, graph rewrite systems can be used to query the graph.
- ▶ Contrary to F-DATALOG and query languages, edge graph rewrite systems materialize their results instantly.
 - Therefore, they are amenable for *model analysis and mappings*
 - Graph rewriting is restricted to binary predicates and always yields all solutions
- ▶ General graph rewriting can do transformation, i.e. is much more powerful than F-DATALOG.
 - Graph rewriting enables a uniform view of the entire optimization process
 - There is no methodology on how to specify general abstract interpretations with graph rewrite systems
 - In interprocedural analysis, instead of chaotic iteration special evaluation strategies must be used [Reps95] [Knoop92]
 - Currently strategies have to be modeled in the rewrite specifications explicitly
- ▶ Uniform Specification of Analysis and Transformation [Aßmann00]
 - If the program analysis (including abstract interpretation) is specified with GRS, it can be unified with program transformation

41.4.1 Implementation of Data-Flow Analysis in Tools



Graph Rewrite Tools for Graph Reachability

- ▶ GrGen graph rewriting system (U Karlsruhe)
 - [Wwww.grgen.net](http://www.grgen.net)
- ▶ Fujaba graph rewrite system www.fujaba.de
- ▶ (e)MOFLON graph rewrite system www.moflon.de
 - TGG for Model Mapping, similar to QVT-R
 - See chapter MOFLON
- ▶ AGG graph rewrite system (From Berlin and Marburg)
 - <http://user.cs.tu-berlin.de/~gragra/agg/>
- ▶ VIATRA2 graph rewrite system on EMF
 - <http://eclipse.org/gmt/VIATRA2/>
- ▶ GROOVE for the construction of interpreters
 - <http://groove.cs.utwente.nl/>

Optimix: using Efficient Evaluation Algorithms from Logic Programming

- ▶ Tool OPTIMIX uses the „Order algorithm“ scheme [Aßmann00]
 - Generates target code of a programming language
 - Code generation uses variants of nested loop join algorithm
 - Works effectively on very sparse directed graphs
 - Bottom-up evaluation, as in F-Datalog; top-down evaluation as in Prolog possible, with resolution
- ▶ Optimizations from Datalog and F-Datalog
 - Bottom-up evaluation is normal, as in Datalog
 - Top-down evaluation as in Prolog possible, with resolution
 - Sometimes fixpoint evaluations can be avoided
 - Use of index structures possible
 - Linear bitvector union operations can be used
 - semi-naive evaluation
 - index structures
 - magic set transformation
 - transitive closure optimizations

41.5 Model Mappings in In-Memory Megamodels (Modellverknüpfung) and Their Use for Traceability

- Model mapping languages are model query languages who enter their results again into the models as analysis information.
- They create *model mappings* which are important for macromodels.

Obligatory Literature

- ▶ [BERS08] Daniel Bildhauer, Jürgen Ebert, Volker Riediger, and Hannes Schwarz. Using the TGraph Approach for Model Fact Repositories. . In: Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008). S. 9--18.
- ▶ Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and System Modeling*, 9 (4):473-492, 2010.

Inter-Model Analysis with Reachability

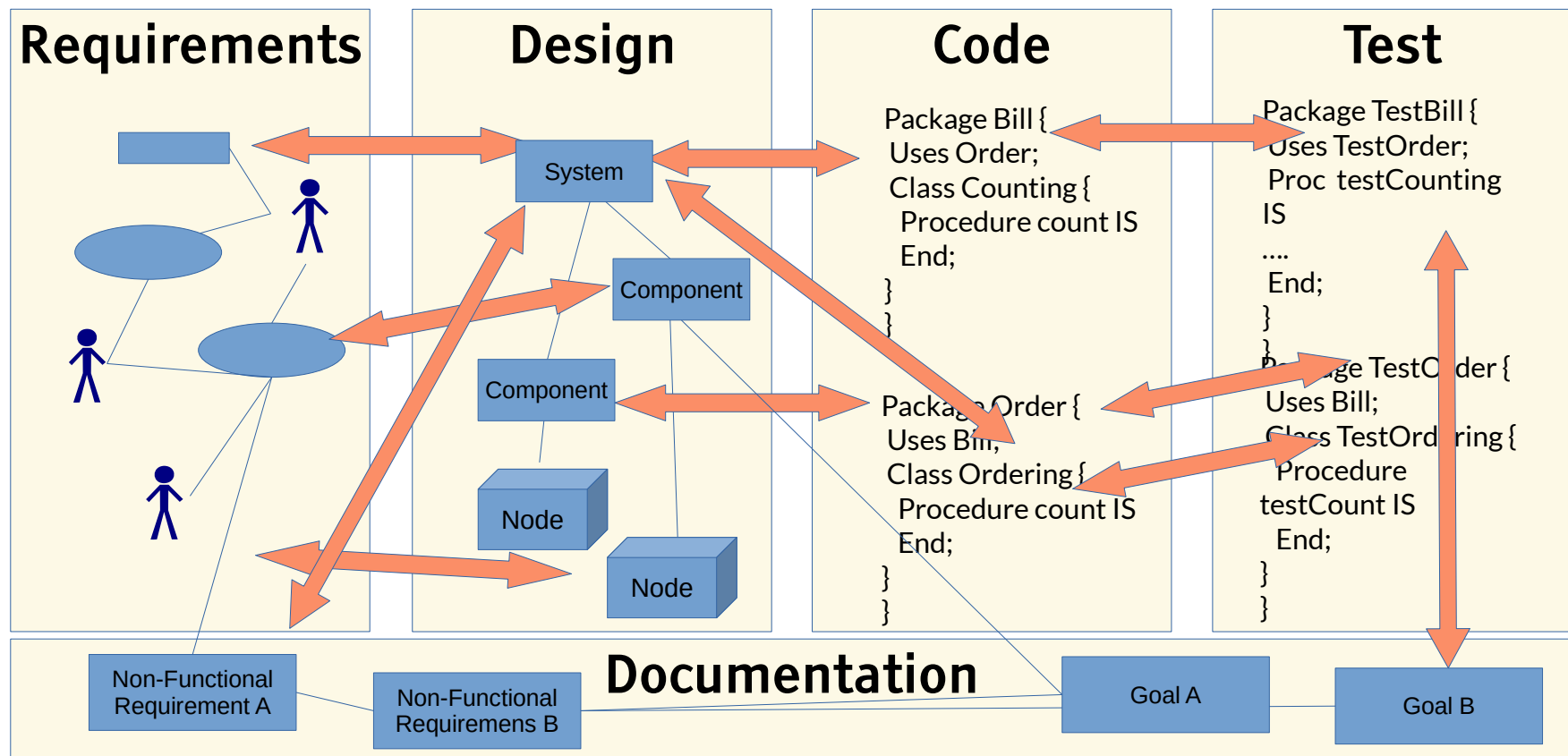
- ▶ **Deep model analysis:** Graph reachability analyzers create direct mappings (graphs) from indirect mappings (abbreviate intensional or recursive mappings)
 - for reachability of model elements
 - to create model slicings (projections to some subgraphs)
 - to prepare refactorings, transformers, and optimizers
 - For models: For model refactoring, adaptation and specialization, weaving and composition
 - For code: Portability to new processor types and memory hierarchies
 - For optimization (time, memory, energy consumption)
- ▶ For **traceability** of model elements in *other models*. Traceability is reachability of model elements over several models

Inter-Model Relationships in The ReDoDeCT Macromodel

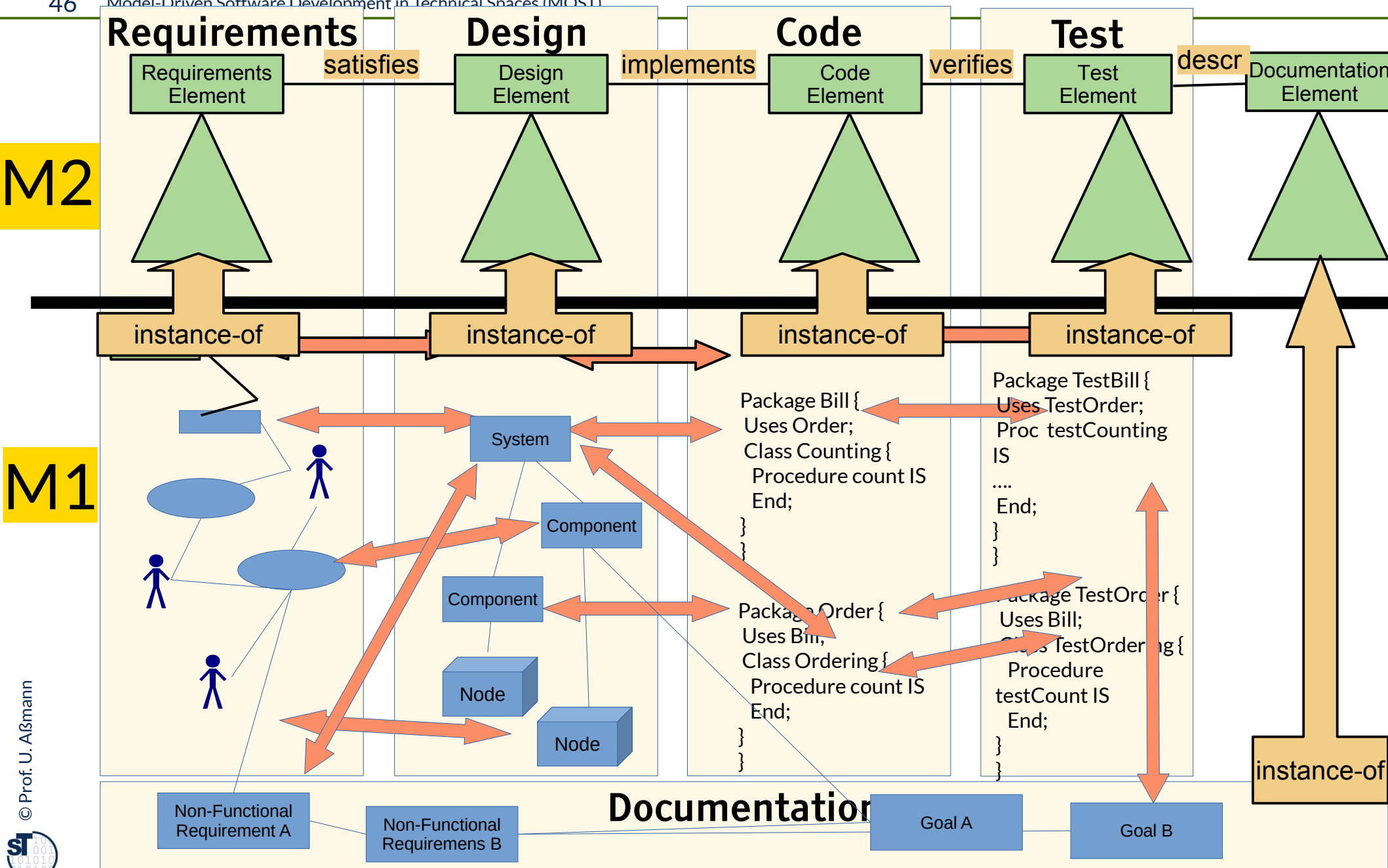
- ▶ An **inter-model relationship** is a relationship between model elements of different models
 - Here: expresses mapping between the Requirements model, Documentation, Design model, Code, Test cases
- ▶ The **ReDoDeCT macromodel** relies on **inter-model relationships** between all 4 models

Q12: The ReDoDeCT Problem and its Macromodel

- ▶ The **ReDoDeCT problem** is the problem how requirements, documentation, design, code, and tests are related (→ V model)
- ▶ Mappings between the Requirements model, Documentation files, Design model, Code, Test cases
- ▶ A **ReDoDeCT macromodel** has maintained mappings between all 5 models



- ▶ An (direct) inter-model relationship is defined between top-level metaclasses in the models of the macromodel
- ▶ The ReDoDeCT macromodel defines on direct inter-model relationships on RequirementsElement, DesignElement, CodeElement, TestElement, DocumentationElement



Specification of Traceability in ReDeCT with GrGen and TGreQL

- ▶ **Direct inter-model relationships** form the basis of queries in the macromodel. Allow for the definition of
 - **Traceability relations** between model elements of different models
 - Hyperedges (tuples) between several model elements of different models
- ▶ **Any query language can be used for model mappings, if their results are entered into the model resp. macromodel**

// GrGen notation:

```
rule collectInterModelDep(r:Req, d:Des, c:Code, t:Test) {  
  r -:reqs-> req:RequirementsElement;  
    req.name="Count Bill";  
  d -:arch-> archElem:DesignElement;  
    archElem -:Satisfies->req;  
  d -:design-> desElem:DesignElement;  
    desElem -:Realize->archElem;  
  c -:has-> class:Class;  
    class -:Implements->desElem;
```

```
// Defining a inter-model hyperedge (tuple) in TGreQL [BERS08]  
elementsIn(  
  from req:V{RequirementsElement}, archElem:V{DesignElement},  
        desElem:V{DesignElement}, class:V{ClassDefinition}  
  with req.name="Count Bill"  
        and req <-- {Satisfies} archElem  
        and archElem <-- {Realize} desElem  
        and desElem <-- {Implements} class  
  report req, archElem, desElem, class  
end  
)
```

The End - Appendix

Comprehension Questions

- ▶ Why do EARS correspond to binary Datalog? why is EARS a similar query language as .QL?
- ▶ Explain program slicing as an application of graph reachability.
- ▶ Why is regular graph reachability “regular”? What is the different to context-free graph reachability?
- ▶ How do you create a model mapping with regular graph reachability?
- ▶ Explain a typical data-flow analysis with EARS. Why do EARS rules that rewrite the information “around” the control-flow graph form an abstract interpreter?
- ▶ EARS can rewrite models. How would you specify a model refactoring engine with EARS?
- ▶ Why are EARS good for traceability in megamodels?

41. Deep Graph Model Analysis and Macromodels: Model and Program Analysis with (Recursive) Graph Reachability

How Context-Sensitive Constraints can be Checked in a Model

Prof. Dr. Uwe Aßmann
Softwaretechnologie
Technische Universität Dresden
Version 21-1.1, 29.01.22

- 1) Graph Reachability as Deep Analysis
 - 1) EARS
 - 1) Regular graph reachability and Slicing
 - 2) Graph slicing
 - 3) Value-flow analysis
 - 1) Context-free graph reachability
 - 4) More on the Graph-Logic Isomorphism
 - 1) Implementation in Tools
 - 5) Model Mappings in Megamodels

- ▶ GrGen web site <http://www.info.uni-karlsruhe.de/software/grgen/>
- ▶ GrGen User Manual
<http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf>
- ▶ [Aßmann00] 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>
- ▶ Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
 - <http://www.springerlink.com/content/5742246115107431/>
- ▶ Thomas Reps. Program analysis via graph reachability. Information and Software Technology, 40(11-12):701-726, November 1998. Special issue on program slicing.
- ▶ Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352-357, July 1984.
- ▶ Frank Tip. A survey of program slicing techniques. Journal of Programming Languages, 3:121-189, 1995.



- ▶ B. Courcelle. Graphs as relational structures: An algebraic and logical approach. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, 4th International Workshop On Graph Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 238-252. Springer, March 1990.
- ▶ B. Courcelle. The logical expression of graph properties (abstract). In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, 4th International Workshop On Graph Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 38-40. Springer, March 1990.
- ▶ B. Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, Handbook of Theoretical Computer Science, pages 193- 242, Amsterdam, 1990. Elsevier Science Publishers.

Other References

- ▶ Uwe Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In Graph Grammar Handbook, Vol. II. Chapman-Hall, 1999.
- ▶ K. Lano. Catalogue of Model Transformations
 - <http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf>

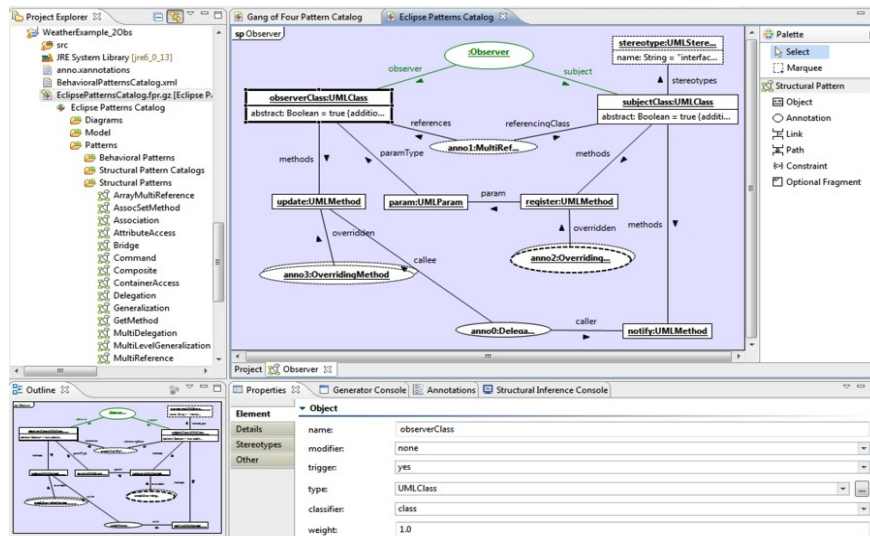


41.1. Introduction to Diagrammatic Storyboard Rule Notation for Graph Rewriting

Originally introduced by Fujaba www.fujaba.de (tool now unsupported)



- ▶ Fujaba is a MetaCASE-tool based on GRS with home-grown metalanguage and metamodel
- ▶ Basic technology: graph pattern matching and rewriting



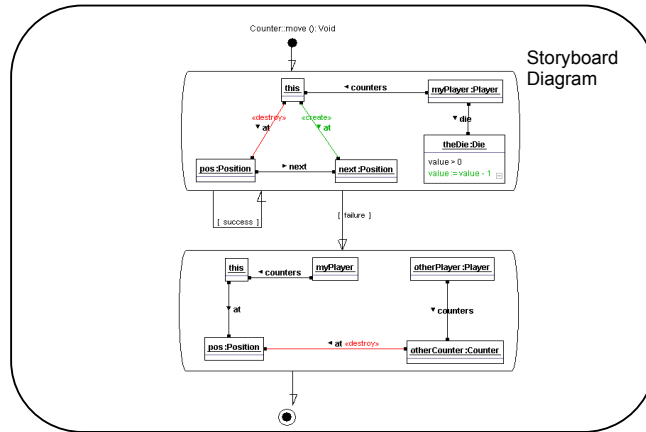
© Prof. U. Altmann



<http://www.fujaba.de/typo3temp/pics/604c5c6c9e.png>

Fujaba Storyboard Diagrams for Adding and Removing Graph Fragments

- ▶ Storyboards are activity diagrams in which activities are GRS (graph notation with colors)
- ▶ Green color: adding model fragments; Red color: deleting them
- ▶ Pool starts at node this and reaches into the object net
- ▶ GRS can be embedded into Petri Nets, DFG and other BSL



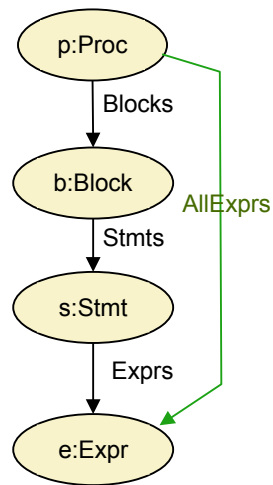
41.1 Using EARS (Binary Edge Addition) for Deep Analysis of Models and Mappings of Models and Code

- and as the Bridge to Graph Rewriting

- Graph reachability engines are analysis tools answering questions about the deeper structure of models and programs
- EARS can be employed for regular graph reachability, context-free graph reachability, slicing, data-flow analysis
- And traceability for inter-model relationships

EARS for Model Mapping

- ▶ **Edge addition rewrite systems (EARS)** compute direct binary relations for remotely reachable parts of a graph and a model
 - They **abbreviate long paths** in models
- ▶ EARS can be used for reachability of elements in models and model mapping:
 - Transitive closure
 - Regular path reachability
 - Context-free path reachability
- ▶ EARS form the bridge to graph rewriting and graph-rewriting based model transformations
- ▶ They correspond to binary Datalog



Model Analysis with Graph Reachability

- ▶ Use the **graph-logic-isomorphism**: Represent everything in a program or a model as directed graphs
 - Program code (control flow, statements, procedures, classes)
 - Model elements (states, transitions, ...)
 - Analysis information (abstract domains, flow info ...)
 - Directed graphs with node and edge types, node attributes, one-edge condition (no multi-graphs)
- ▶ **Use edge decomposition as textual notation**
- ▶ Use edge addition rewrite systems (EARS), Datalog and other graph reachability specification languages to
 - Query the graphs (on values and patterns)
 - Analyze the graphs (on reachability of nodes)
 - Map the graphs to each other (model mapping)
- ▶ Later: Use graph rewrite systems (GRS) to construct and augment the graphs, transform the graphs

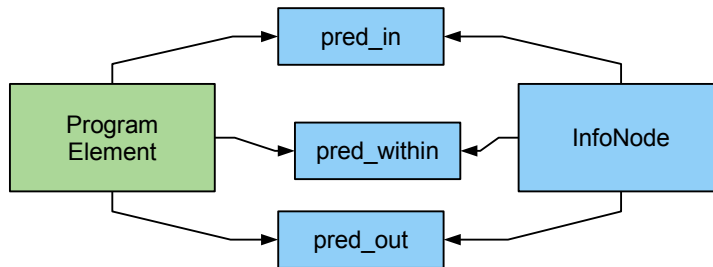


Specification Process with Graph Rewrite Systems and EARS

- 1) Specification of the data model (graph schema) with a graph-like DDL (ERD, MOF, GXL, UML or similar):**
 - **Schema of the program representation:** program code as objects and basic relationships. This data, i.e., the start graph, is provided as result of the parser
 - **Schema of analysis information** (the inferred predicates over the program objects) as objects or relationships
- 2) Flat model and program analysis** (preparing the abstract interpretation)
 - Querying graphs, enlarging graphs, static slicing, Reachability
 - Equivalence classing
 - Materializing implicit knowledge to explicit knowledge
- 3) Deep model and program analysis**
 - Inter-model reachability (traceability), materializing model mappings
 - **Abstract Interpretation** (program analysis as interpretation)
 - Specifying the transfer functions of an abstract interpretation of the program with graph rewrite rules on the analysis information
- 4) Model and Program transformation** Transforming the program representation
 - Optimization such as peephole optimization or constant folding (context-free)
 - Code motion (Context-sensitive)

Deep Analysis and Abstract Interpretation with Graph Rewriting

- ▶ A *graph-rewriting based abstract interpreter* stores, for every program element of the program graph (Expr, Stmt, Block, Proc, Class < ProgramElement) three “truths” (values) for every node in the analysis information (InfoNode):
 - `p:ProgramElement -.:predicate_in-> i:InfoNode`
 - `// predicate_in(p,i)`
 - `p:ProgramElement -.:predicate_within-> i:InfoNode`
 - `// predicate_within(p,i)`
 - `p:ProgramElement -.:predicate_out-> i:InfoNode`
 - `// predicate_out(p,i)`
- ▶ Values of program elements are encoded as an edge between program elements and InfoNodes





41.2. Reachability of Model Elements and Models for Model Analysis and Mapping

- ▶ With model mapping languages, such as edge addition rewrite systems or TGreQL





41.2.1. Simple Reachability of Model Elements and Models: Path Abbreviations in Graph Analysis

- ▶ With model mapping languages, such as edge addition rewrite systems or TGreQL

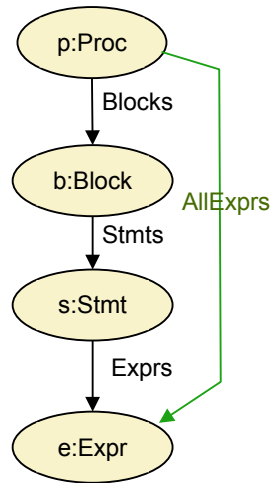


Path Abbreviations for Simple Reachability

- ▶ Path abbreviations shorten paths in the manipulated graph.
- ▶ They may collect nodes into the neighborhood of other nodes.
- ▶ Ex.: Collection of Expressions for a procedure: edge addition

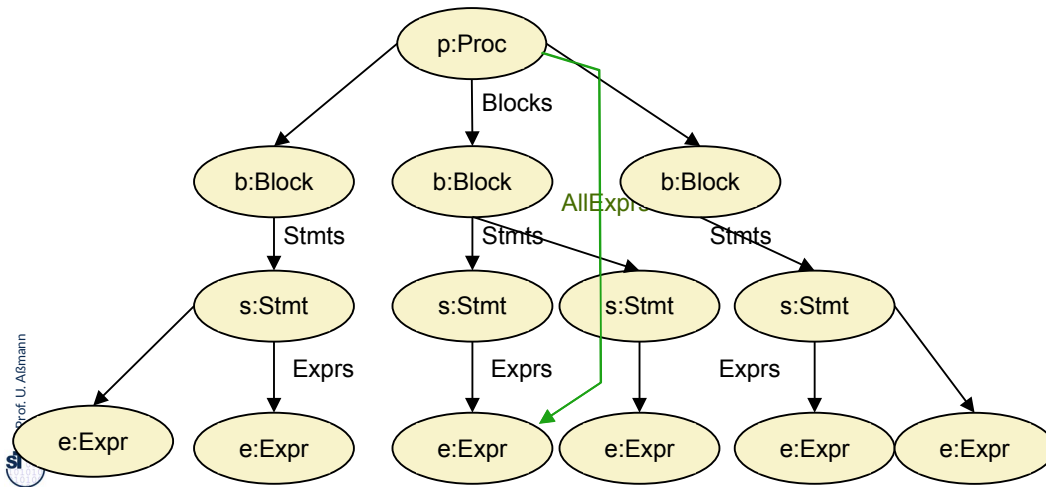
```
-- GrGen notation:  
rule collectAllExpr(p:Proc) {  
  p --Blocks-> b:Block;  
  b --Stmts-> s:Stmt;  
  s --Exprs-> e:Expr;  
  modify {  
    p --AllExprs-> e;  
  }  
}
```

```
-- F-DataLog notation (edge decomposition):  
AllExprs(Proc, Expr) :-  
  Blocks(Proc, Block),  
  Stmts(Block, Stmt),  
  Exprs(Stmt, Expr).  
-- if-then rules:  
if Blocks(Proc, Block),  
  Stmts(Block, Stmt),  
  Exprs(Stmt, Expr)  
then  
  AllExprs(Proc, Expr);  
- regular expression notation (TGreQL):  
AllExprs := Proc Blocks.Stmts.Expr Expr
```



Forward Slicing from a Point in the ProgramGraph (Single-Source Multiple-Target (SSMT) Problems)

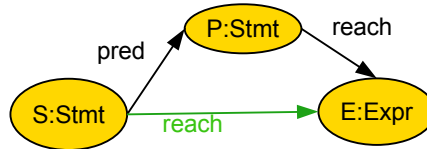
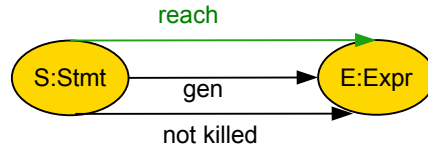
- ▶ A *forward slice* (SSMT-region) has one source, many targets and all intermediate nodes
- ▶ The slice border is the border of the region



Linear Recursion for Remote Reachability



- ▶ Reachability most often can be reduced to transitive closure of one or several relations.
- ▶ **Query: "Does an Stmt S reach a expression E?"**
- ▶ TC combines path abbreviation with recursion
 - F-Datalog, GrGen: Left or right recursion
 - Kleene * in TGreQL
 - Thick arrow in Fujaba



```

// TGreQL
reach[gen]*(S:Stmt,E:Expr)

// GrGen can use inheritance on
// nodes and edges
rule reachability (s:Node) {
  s -:BasicEdge-> p:Node;
  p -:RecursiveEdge-> e:Node;
  modify {
    s -:RecursiveEdge-> e:Node
  }
}
  
```

© Prof. U. Altmann

```

// F-Datalog
reach(S:Stmt,E:Expr) :- gen(S:Stmt,E:Expr), not killed(S:Stmt,E:Expr).
reach(S:Stmt,E:Expr) :- pred(S:Stmt,P), reach(P,E:Expr).
  
```

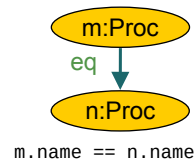
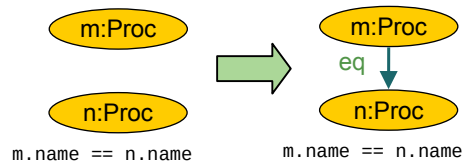

Ex.: Relating Nodes into Equivalence Classes

- ▶ Ex.: Computing equivalent nodes
- ▶ Context-sensitive problem, because m is not in the context of n

```
F-Datalog baserule:  
eq(m:Proc,n:Proc) :-  
  m.name == n.name.  
-- If-then:  
If (m:Proc, n:Proc) and m.name == n.name)  
  eq(m,n)  
}  
- TgreQL regular expression:  
m:Proc eq n:Proc if  
m.name == n.name
```

```
// GrGen  
rule buildGraph(m:Node, n:Node) {  
  m.Name == n.Name;  
  modify { m:-eq-> n }  
}
```

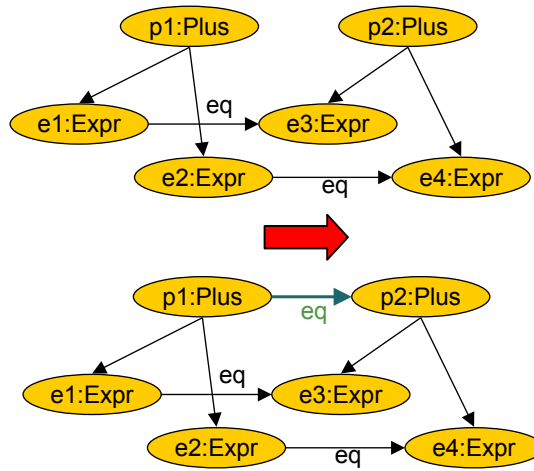
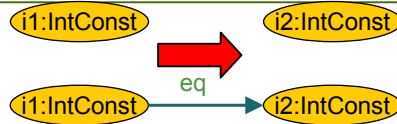
© Prof. U. Altmann



Ex. Relating Nodes into Equivalence Classes (Here: Value Numbering, Synt. Expression Equivalence)

- ▶ Ex.: Computing structurally equivalent expressions with bi-recursive reachability
- ▶ Question: "Which expression trees have the same structure?"

```
--- F-Datalog baserule:  
eq(i1:IntConst,i2:IntConst) :-  
  i1 == IntConst(Value),  
  i2 == IntConst(Value).  
--- recursive_rule:  
eq(p1:Plus,p2:Plus) :-  
  p1 == Plus(Type),  
  p2 == Plus(Type),  
  Left(p1,e1),  
  Right(p1,e2),  
  Left(p2,e3),  
  Right(p2,e4),  
  eq(e1,e3),  
  eq(e2,e4).
```



41.3. Deep Model Analysis (Value-Flow Analysis, Data-Flow Analysis) as General (Recursive) Graph Reachability over Values

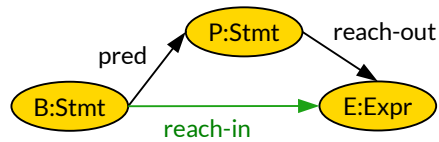
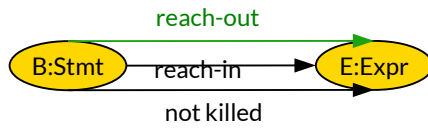
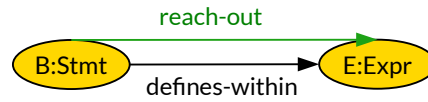
- with edge addition rewrite systems and F-Datalog

- ▶ **Value-flow analysis (data-flow analysis)** is a specific form of deep model analysis asking **reachability questions**, i.e., computing the *flow of data (value flow)* through the model or program, from variable assignments to variable uses
 - Result: the **value-flow graph (data-flow graph)**
 - If the value flow analysis is done along the control-flow graph, it is called an **abstract interpretation** of a program
 - EARS can do an abstract interpretation of a program, if they are rewriting on the control-flow graph. Then, their rules implement transfer functions of an abstract interpreter
- ▶ Examples of reachability problems:
 - **AllSuperClasses**: find out for a class transitively all superclasses
 - **AllEnclosingScopes**: find out for a scope all enclosing scopes
 - **AllEnclosingWholes**: find out, for a part, its wholes into which it is included
 - **Reaching Definitions Analysis**: Which Assignments (Definitions) of a variable can reach which statement?
 - **Live Variable Analysis**: At which statement is a variable live, will further be used?
 - **Busy Expression Analysis**: Which expression will be used on all outgoing paths?



Reaching Definition Analysis By Abstract Interpretation with EARS (Reachable Statements from Expression Definition)

- ▶ **Query:** "Which definitions of expressions reach which statement?"
 - Assignments of a variable, temporary, or register
 - Usually computed for all positions *before* and *after* a statement
- ▶ Graph rewrite rules implement an abstract interpreter
 - On instructions or on blocks of instructions
 - Flow information is expressed with edges of relations "reach-*"
- ▶ Recursive system (via edge reach-in)
 - (B reach-out E) := (E reaches end of block B)
- ▶ GrGen can express this via its generic reachability rules



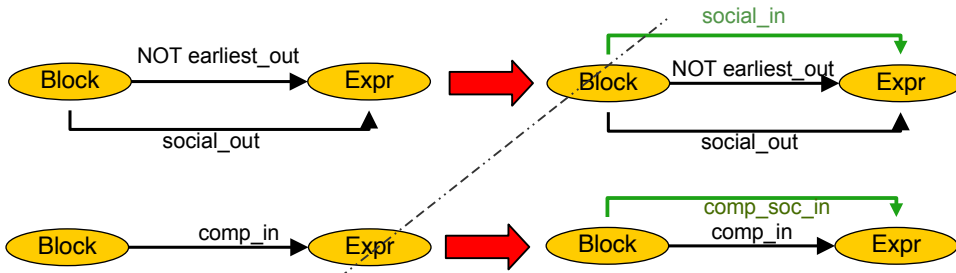
```
reach-out(B,E) :- defines-within(B,E).
reach-out(B,E) :- reach-in(B,E), not killed(B,E).
reach-in(B,E)  :- pred(B,P), reach-out(P,E).
```

- ▶ **Code motion** is an essential transformation to speed up the generated code. However, it is a complex transformation:
 - Discovering loop-invariant expressions by data-flow analysis
 - Moving loop-invariant expressions out of loops upward
 - Code motion needs complex data-flow analysis
- ▶ **Busy Code Motion (BCM)** moves expressions as upward (early) as possible
- ▶ **Lazy Code Motion (LCM)**
 - Moving expressions out of loops to the front of the loop, upward, but carefully:
 - Moving expressions to an optimal place so that register lifetimes are shorter and not too long (optimally early)
 - LCM analysis computes this optimal early place of an expression [Knoop/Steffen]
 - Analyze an optimally early place for the placement of an expression
 - About 6 equation systems similar to reaching-definitions
 - Every equation system is an EARS [Aßmann00]

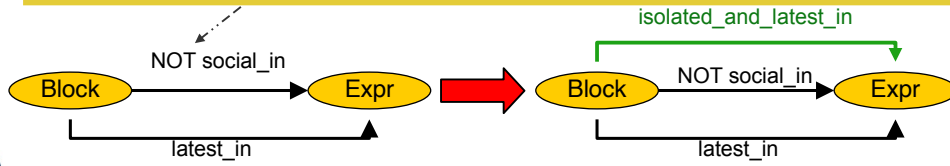
Excerpt from LCM Analysis with Overlaps

► Compute an optimally early block for an expression (out of a loop)

Query: "Which expression is not isolated (social) at the beginning of a block?"



Query: "Which expression is not isolated (social) at the beginning of a block?"





41.3.2 Regular Graph Reachability and Slicing

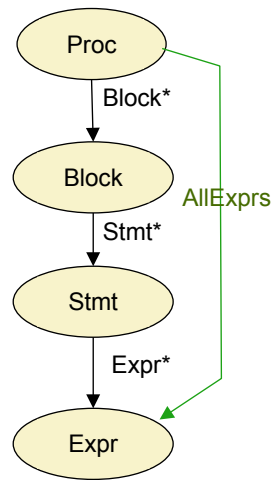


Regular Graph Reachability

- ▶ If the query can be expressed as a regular expression, the query is a **regular graph reachability problem**
- ▶ Kleene star is used as transitive closure operator
- ▶ TqreQL and Fujaba are languages offering Kleene *

```
-- F-Datalog notation:
AllExprs(Proc, Expr) :-
  Block*(Proc, Block),
  Stmt*(Block, Stmt),
  Expr*(Stmt, Expr).
-- GrGen if-then rules:
if Proc -:Block*-> Block,
   Block -:Stmt*-> Stmt,
   Stmt -:Expr*-> Expr
modify {
  Proc -:AllExprs-> Expr
}
-- regular expression notation (TGreQL):
AllExprs := Proc Block*.Stmt*.Expr* Expr
```

© Prof. U. Altmann



Static Slicing: Single-Source-Multiple-Target Regular Reachability (Regular Reachable Dependencies)

- ▶ [Weiser] [Tip]
- ▶ A **static slice** is the region of a program or model *dependent* from *one source* node (reachable by a regular reachability query in a dependency graph)
 - A static slice is a *single-source path regular reachability problem (SSPP)* on the dependency graph
 - A static slice introduces path abbreviations from one entity to a region
- ▶ A **forward slice** is a dependent region in *forward* direction of the program
 - The uses of a variable
 - The callees of a call
 - The uses of a type
- ▶ A **backward slice** is a dependent region in *backward* direction of the program
 - The assignments which can influence the value of a variable
 - The callers of a method
 - The type of a variable
- ▶ Slicing can map arbitrary entities in programs and models to other entities, based on a regular graph expression



Reachability within Models and Traceability between Models

- ▶ Data-flow analysis (graph reachability, slicing) can be done
 - Intraprocedurally (within one procedure)
 - Interprocedurally (program-wide)
- ▶ **Traceability** is inter-model slicing and graph reachability
 - inter-model: then it creates **trace relations** between requirements models, design models, and code models
 - Intra-megamodel: trace relations can trace dependencies between all models in a megamodel, e.g., in an MDA
- ▶ A **model mapping** is an inter-model trace(-ability) graph
 - Model mappings are very important for the dependency analysis and traceability in megamodels and the construction of macromodels



41.3.3 Context-Free Graph Reachability

- ▶ If arbitrary recursion patterns are allowed in F-Datalog and EARS queries, we arrive at context-free graph reachability.



- ▶ Transitive closure and regular graph reachability rely on regular recursion (linear recursion) expressible with the Kleene-* on relations
- ▶ Beyond that, F-Datalog and EARS can describe other recursions
 - Context-free recursions
 - Cross-recursions
- ▶ Then, we speak of **context-free graph reachability**
 - A context-free language describes graph reachability
- ▶ Applications:
 - Complex intraprocedural value flow analyses
 - Interprocedural, whole-program analysis
 - Interprocedural IDFS framework (Reps)
 - Model mappings in a megamodel



41.4 More on the Logic-Graph Isomorphism

- ▶ [Courcelle] discovered that many problems can be expressed in logic (on facts) and in graph rewriting (on graphs)

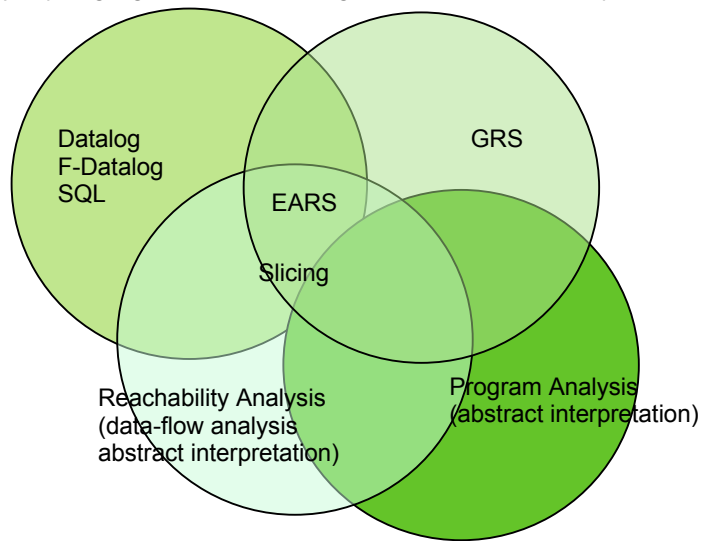


Program and Model Analyses Covered by Graph Reachability

- ▶ Graph Reachability Analysis can do abstract interpretation
 - If it adds analysis information to the program elements and their control-flow graph
 - Slicing is a Single-Source-Multiple-Target reachability analysis
- ▶ Every abstract interpretation where a mapping of the abstract domains to graphs can be found.
 - Monotone and distributive data-flow analysis
Control flow analysis (and callee analysis)
 - Static-single-assignment (SSA) construction
 - Interprocedural IDFS analysis framework (Reps)

The Common Core of Logic, Graph Rewriting and Program Analysis

- ▶ Graph rewriting, DATALOG and data-flow analysis have a common core: EARS
- ▶ Datalog query languages such as .QL or TgreQL can be extended by GRS



- ▶ Abstract interpretation (Data-flow analysis), F-DATALOG and graph rewrite systems have a common kernel: EARS
 - As F-DATALOG, graph rewrite systems can be used to query the graph.
- ▶ Contrary to F-DATALOG and query languages, edge graph rewrite systems materialize their results instantly.
 - Therefore, they are amenable for *model analysis and mappings*
 - Graph rewriting is restricted to binary predicates and always yields all solutions
- ▶ General graph rewriting can do transformation, i.e. is much more powerful than F-DATALOG.
 - Graph rewriting enables a uniform view of the entire optimization process
 - There is no methodology on how to specify general abstract interpretations with graph rewrite systems
 - In interprocedural analysis, instead of chaotic iteration special evaluation strategies must be used [Reps95] [Knoop92]
 - Currently strategies have to be modeled in the rewrite specifications explicitly
- ▶ Uniform Specification of Analysis and Transformation [Aßmann00]
 - If the program analysis (including abstract interpretation) is specified with GRS, it can be unified with program transformation



41.4.1 Implementation of Data-Flow Analysis in Tools



Graph Rewrite Tools for Graph Reachability

- ▶ GrGen graph rewriting system (U Karlsruhe)
 - [Www.grgen.net](http://www.grgen.net)
- ▶ Fujaba graph rewrite system www.fujaba.de
- ▶ (e)MOFLON graph rewrite system www.moflon.de
 - TGG for Model Mapping, similar to QVT-R
 - See chapter MOFLON
- ▶ AGG graph rewrite system (From Berlin and Marburg)
 - <http://user.cs.tu-berlin.de/~gragra/agg/>
- ▶ VIATRA2 graph rewrite system on EMF
 - <http://eclipse.org/gmt/VIATRA2/>
- ▶ GROOVE for the construction of interpreters
 - <http://groove.cs.utwente.nl/>

Optimix: using Efficient Evaluation Algorithms from Logic Programming

- ▶ Tool OPTIMIX uses the „Order algorithm“ scheme [Aßmann00]
 - Generates target code of a programming language
 - Code generation uses variants of nested loop join algorithm
 - Works effectively on very sparse directed graphs
 - Bottom-up evaluation, as in F-Datalog; top-down evaluation as in Prolog possible, with resolution
- ▶ Optimizations from Datalog and F-Datalog
 - Bottom-up evaluation is normal, as in Datalog
 - Top-down evaluation as in Prolog possible, with resolution
 - Sometimes fixpoint evaluations can be avoided
 - Use of index structures possible
 - Linear bitvector union operations can be used
 - semi-naive evaluation
 - index structures
 - magic set transformation
 - transitive closure optimizations



41.5 Model Mappings in In-Memory Megamodels (Modellverknüpfung) and Their Use for Traceability

- Model mapping languages are model query languages who enter their results again into the models as analysis information.
- They create *model mappings* which are important for macromodels.

Obligatory Literature

- ▶ [BERS08] Daniel Bildhauer, Jürgen Ebert, Volker Riediger, and Hannes Schwarz. Using the TGraph Approach for Model Fact Repositories. . In: Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008). S. 9--18.
- ▶ Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and System Modeling*, 9 (4):473-492, 2010.

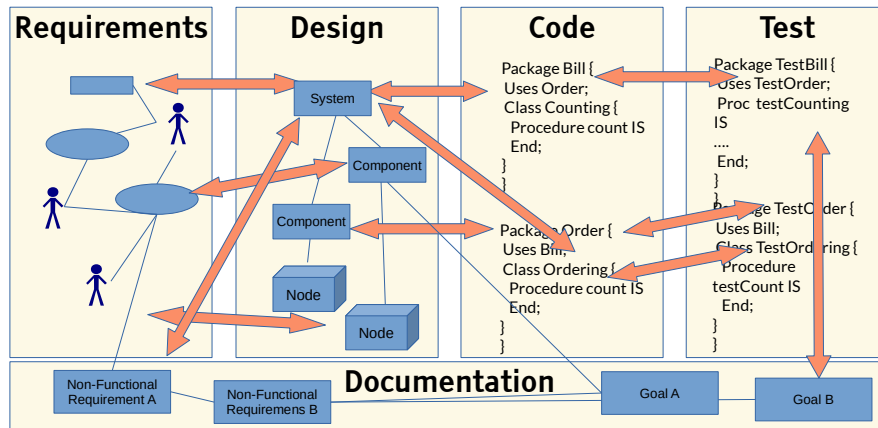
- ▶ **Deep model analysis:** Graph reachability analyzers create direct mappings (graphs) from indirect mappings (abbreviate intensional or recursive mappings)
 - for reachability of model elements
 - to create model slicings (projections to some subgraphs)
 - to prepare refactorings, transformers, and optimizers
 - For models: For model refactoring, adaptation and specialization, weaving and composition
 - For code: Portability to new processor types and memory hierarchies
 - For optimization (time, memory, energy consumption)
- ▶ For **traceability** of model elements in *other models*. Traceability is reachability of model elements over several models

Inter-Model Relationships in The ReDoDeCT Macromodel

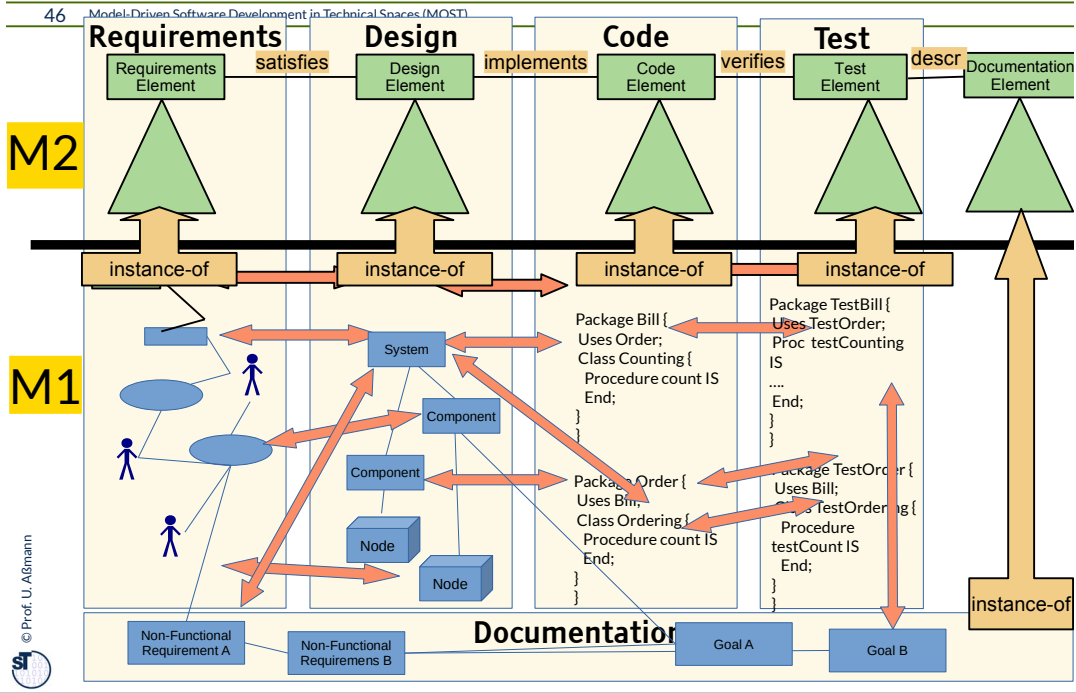
- ▶ An **inter-model relationship** is a relationship between model elements of different models
 - Here: expresses mapping between the Requirements model, Documentation, Design model, Code, Test cases
- ▶ The **ReDoDeCT macromodel** relies on **inter-model relationships** between all 4 models

Q12: The ReDoDeCT Problem and its Macromodel

- ▶ The **ReDoDeCT problem** is the problem how requirements, documentation, design, code, and tests are related (→ V model)
- ▶ Mappings between the Requirements model, Documentation files, Design model, Code, Test cases
- ▶ A **ReDoDeCT macromodel** has maintained mappings between all 5 models



- ▶ An (direct) inter-model relationship is defined between top-level metaclasses in the models of the macromodel
- ▶ The ReDoDeCT macromodel defines on direct inter-model relationships on RequirementsElement, DesignElement, CodeElement, TestElement, DocumentationElement



Specification of Traceability in ReDeCT with GrGen and TGreQL

- ▶ **Direct inter-model relationships** form the basis of queries in the macromodel. Allow for the definition of
 - **Traceability relations** between model elements of different models
 - Hyperedges (tuples) between several model elements of different models
- ▶ **Any query language can be used for model mappings, if their results are entered into the model resp. macromodel**

// GrGen notation:

```
rule collectInterModelDep(r:Req, d:Des, c:Code, t:Test) {  
  r -:reqs-> req:RequirementsElement;  
    req.name="Count Bill";  
  d -:arch-> archElem:DesignElement;  
    archElem -:Satisfies->req;  
  d -:design-> desElem:DesignElement;  
    desElem -:Realize->archElem;  
  c -:has-> class:Class;  
    class -:Implements->desElem;  
}
```

// Defining a inter-model hyperedge (tuple) in TGreQL [BERS08]

```
elementsIn(  
  from req:V{RequirementsElement}, archElem:V{DesignElement},  
        desElem:V{DesignElement}, class:V{ClassDefinition}  
  with req.name="Count Bill"  
        and req <-- {Satisfies} archElem  
        and archElem <-- {Realize} desElem  
        and desElem <-- {Implements} class  
  report req, archElem, desElem, class  
end  
)
```

The End - Appendix Comprehension Questions

- ▶ Why do EARS correspond to binary Datalog? why is EARS a similar query language as .QL?
- ▶ Explain program slicing as an application of graph reachability.
- ▶ Why is regular graph reachability “regular”? What is the different to context-free graph reachability?
- ▶ How do you create a model mapping with regular graph reachability?
- ▶ Explain a typical data-flow analysis with EARS. Why do EARS rules that rewrite the information “around” the control-flow graph form an abstract interpreter?
- ▶ EARS can rewrite models. How would you specify a model refactoring engine with EARS?
- ▶ Why are EARS good for traceability in megamodels?