42. GraphWare: Languages for Graph Transformations and Rewriting

Refactoring, Improvement of Large Models

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 21-0.2, 29.01.22

1) Graph rewriting
2) Complex local graph rewritings
3) Context-sensitive graph rewritings
4) GrGen
5) More on the Graph-Logic Isomorphism

Obligatory Literature

- Kevin Lano. Catalogue of Model Transformations
  - http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf
- 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
- Jakumeit, E., Buchwald, S. & Kroll, M. GrGen.NET. Int J Softw Tools Technol Transfer 12, 263–271 (2010). https://doi.org/10.1007/s10009-010-0148-8
- [GrGenManual] e. Jakumeit, J. Blomer, R. Geiß. The GrGen.NET User Manual Refers to GrGen.NET Release 6.1.1.
  - www.grgen.net
- Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/

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

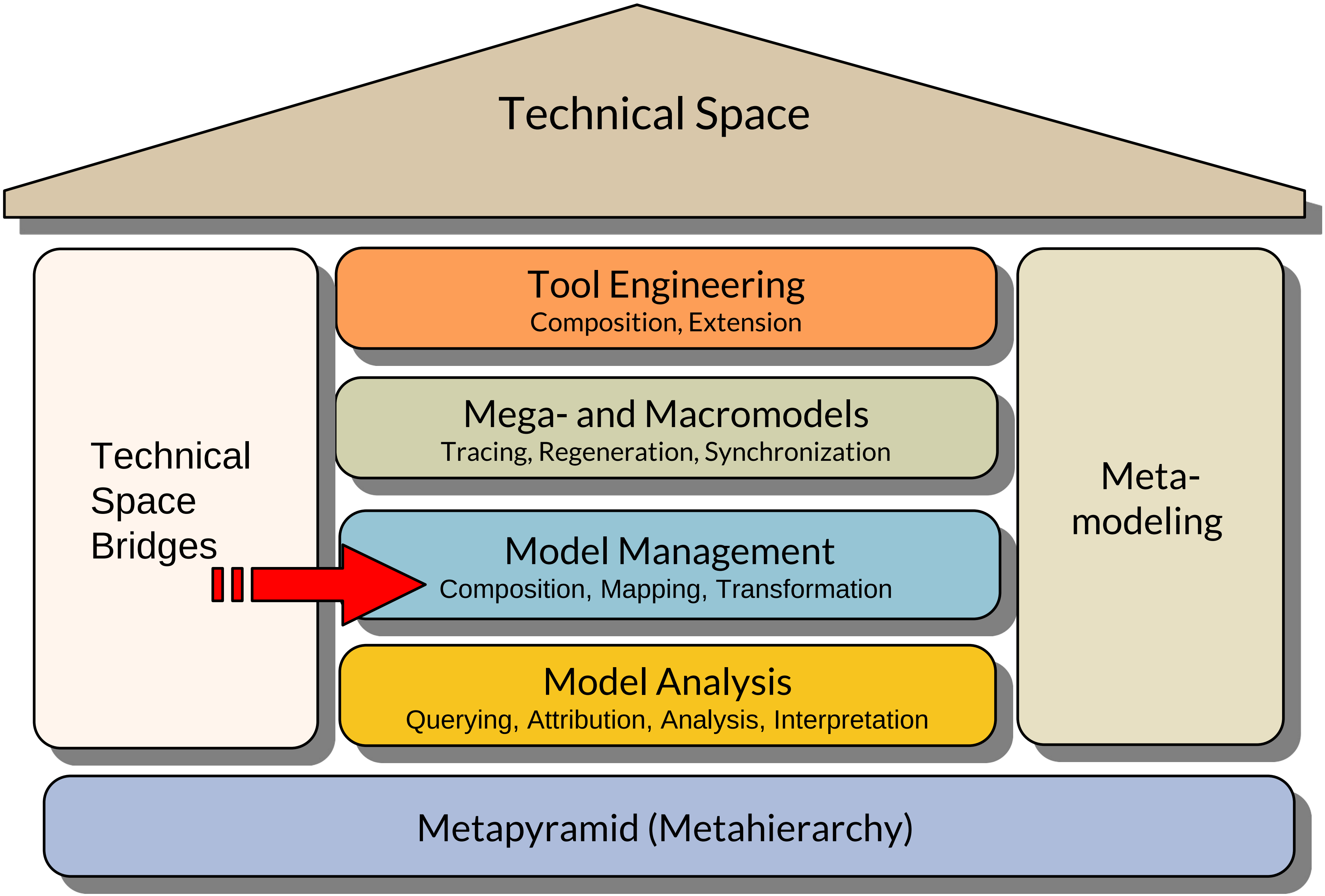
Other Literature

- Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006), pages 1188—1195.
  - http://atlanmod.emn.fr/bibliography/SAC06a
- Tutorial über ATL "Families2Persones"
  - http://www.eclipse.org/m2m/atl/doc/ATLUseCase_Families2Persons.ppt
- ATL Zoo von Beispielen
  - http://www.eclipse.org/m2m/atl/atlTransformations
- A Comparison of ATL and Story-Driven Modeling (Fujaba-style GRS)

http://www.es.tu-darmstadt.de/fileadmin/download/publications/spatzina/PP_AGTIVE_2011.pdf

- Implementation in ATL
  - http://www.eclipse.org/m2m/atl/atlTransformations/EquivalenceAttributesAssociations/EquivalenceAttributesAssociations.pdf

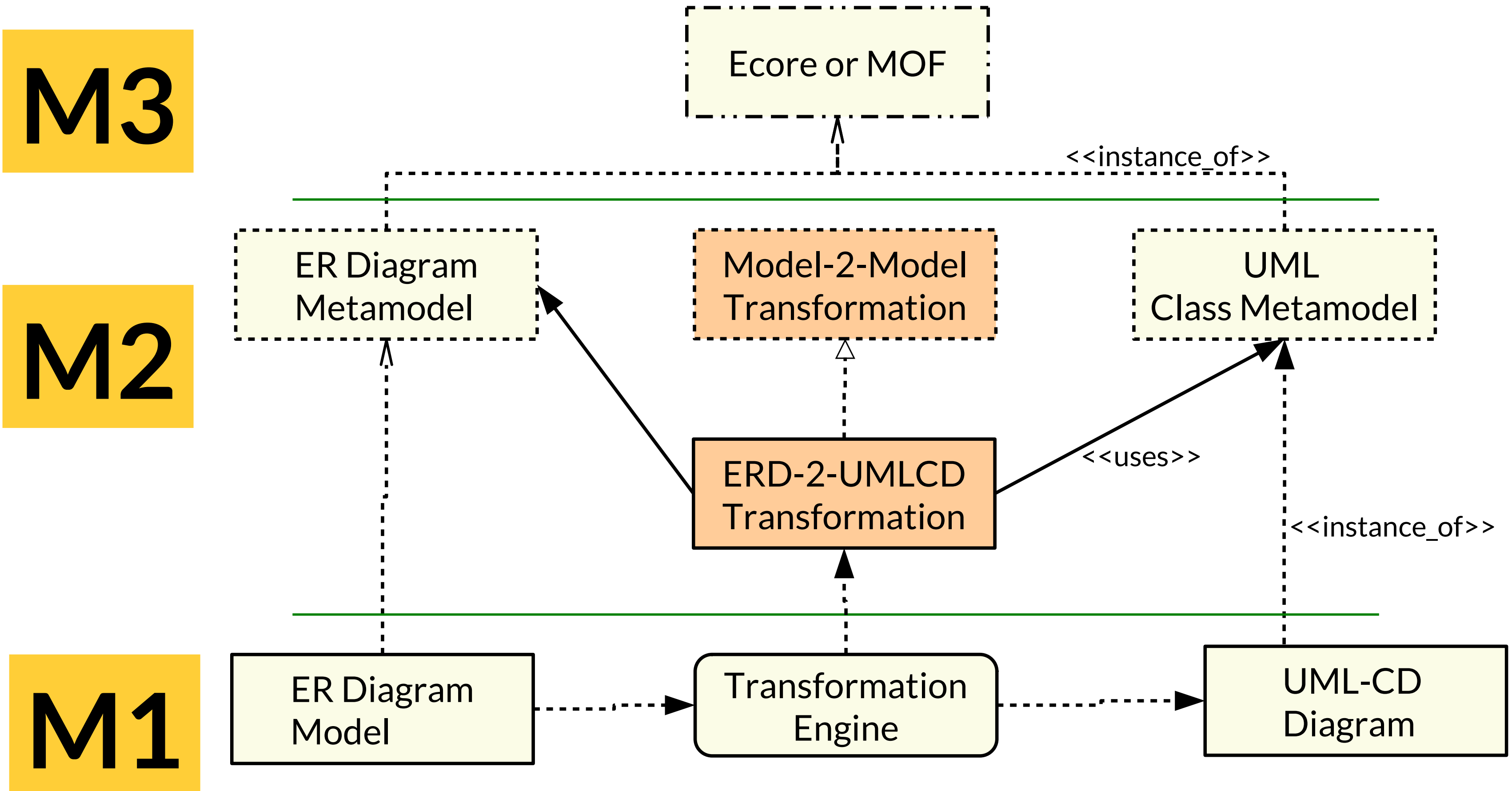
Q10: The House of a Technical Space



42.1. Graph Rewriting for Code and Model Creation, Transformations, Translation

Model Transformations and Their Metamodels

- Model transformations defined in Layer M_{i+1} specify how to transform models on M_i
    - Source and target metamodel are connected by a metamodel of a rule-based transformation language
- **Benefit**: Transformation can be reused for all models, which are instances of the source meta-model



Needs of Graph Rewriting

- M3: Graph-enabling Metalanguage (Ecore, MOF, ERD, RDFS, OWL)
- M2: Metamodel for Nodes and Edges (DDL)
  - MOF, EMOF, tool-specific, etc.
- M2: Metamodel for Rule Language
  - LR Form (left to right rule)
  - Fujaba Storyboard Notation
  - GrGen metamodel
- M1: Node and edge allocators (factories)
  - Graph libraries such as Jgrapht.org
- M0: graph pools with nodes and edges
  - To limit rewriting

Applications of Graph Rewrite Systems for *Transformations* (Graphersetzungssysteme)

- **Concrete and abstract Interpretation** of code and models [Rensink]
- MDSD tools need model transformations
  - Model transformations (Alexander Christoph)
  - Model aspect weaving (Aßmann, Heidenreich, many others)
  - Creation of more specific models in MDA, including the computation of trace create links [Taentzer]
  - Refinement in design [Lano, Schürr, Lewerentz]
  - Refactoring [Mens]
- Compilation and Translation of code and models [Nagl, Aßmann]
- Analysis [Tip, Reps]
  - Slicing
  - Interprocedural analysis with graph reachability
- Optimization [Aßmann]
  - Global code transformations, such as lazy and busy code motion (loop invariant code motion)
- Configuration management [Westfechtel]

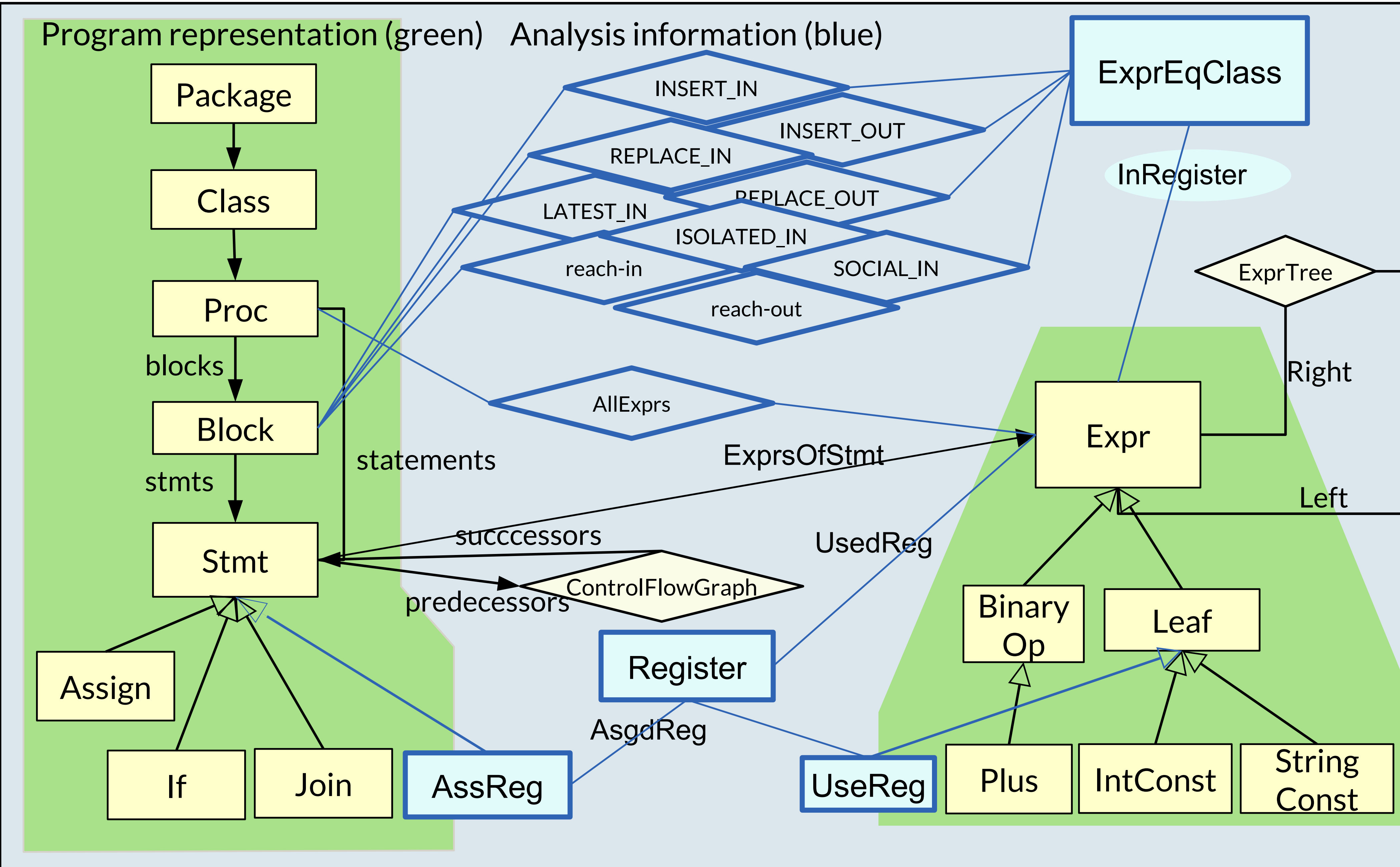
Model Transformation and Optimization with Graph Rewriting

- Use the **graph-logic-isomorphism** [Courcelle]: 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 addition rewrite systems (EARS) to query, analyze, map the graphs to each other
- Use graph rewrite systems (GRS) to create, construct, and augment the graphs
  - Transform the graphs
  - Generate code
- Preferably, the GRS should terminate (XGRS, exhaustive GRS)

Specification Process For Transformation with Deep Analysis

1) Specification of the data model (graph schema)
   - Specification of the 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 infered predicates over the program objects) as objects or relationships
2) „Flat" program analysis (preparing the abstract interpretation)
   - Querying graphs, enlarging graphs
   - Materializing implicit knowledge to explicit knowledge
3) „Deep Analysis": 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) Program transformation (optimization)
   - Transforming the program representation

Q14: A Simple Program Code&Model Schema in MOF



42.2. Complex Local Graph Rewritings

- On Dags (with joins) and Graphs (with cycles)

Code Optimizations Expressible by Local Graph Rewritings

- Local transformations of the program representation
  - copy propagation (copy of expression is loaded to register and reused)
  - constant propagation (constants instead of expressions)
  - branch optimization
  - loop optimizations (unrolling etc.)
  - strength reduction (Multiplications to additions)
  - idiom recognition (pattern matching of complex patterns)
  - dead code elimination (elimination of non-reachable code)

Model Transformations Expressible by Graph Rewritings

[Lano]

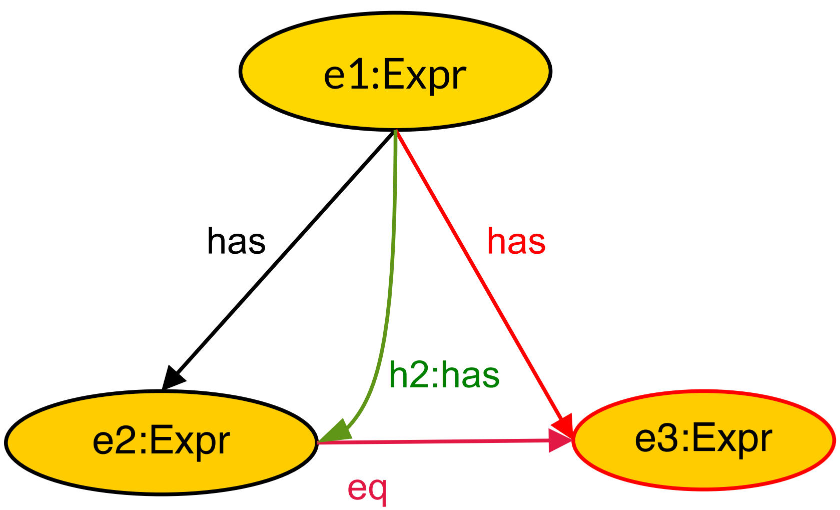
- Transformations of the inheritance hierarchy:
  - Flattening (Reachability)
  - Removal of redundant inheritances
  - Refactorings
    - Split classes
    - Merge classes
    - Move class
- Support during refinement
  - Flattening aggregation, composition, multiple inheritance
- Optimizations:
  - Peephole optimizations (local transformations)
- Generation of dependent models
  - Export to exchange file formats, such as JSON and XML

Example: Peephole Transformation "Local Sharing of Equivalent Subexpressions"


```
if has(Expr1, Expr2),
   has(Expr1, Expr3),
   eq(Expr2,Expr3)
then
   delete Expr3;
   h2:has(Expr1,Expr3)
;
```

- Share common subexpressions
- Here e2

```
// GrGen
rule foldCommonSubexpression(e1:Expr) {
   e1 -:has-> e2:Expr;
   e1 -h1:has-> e3:Expr;
   e2 -e:eq->e3;
   modify { e1:-h2:has-> e2 ; delete(e3);
   delete_edge(h1); delete_edge (e);
 }
}
```

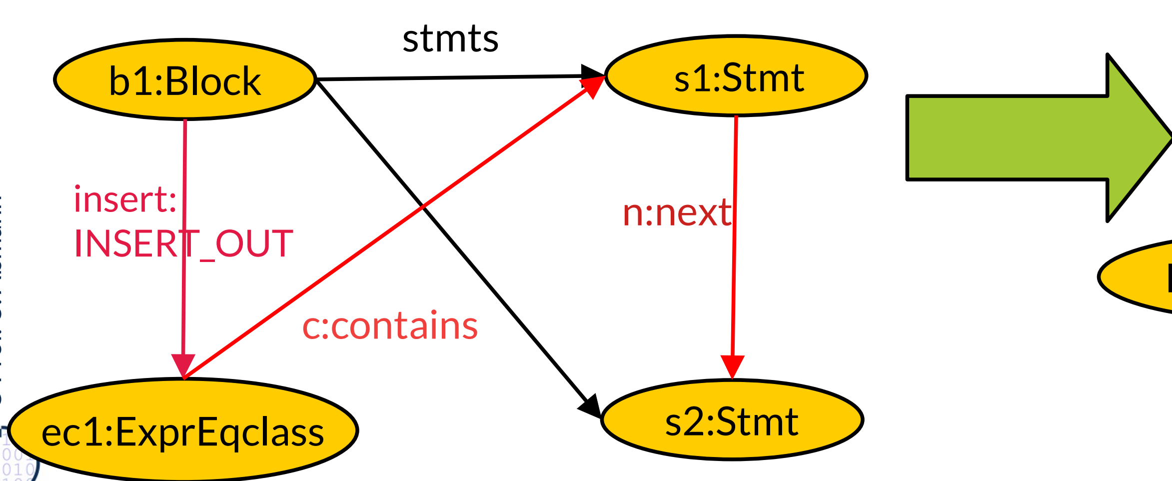


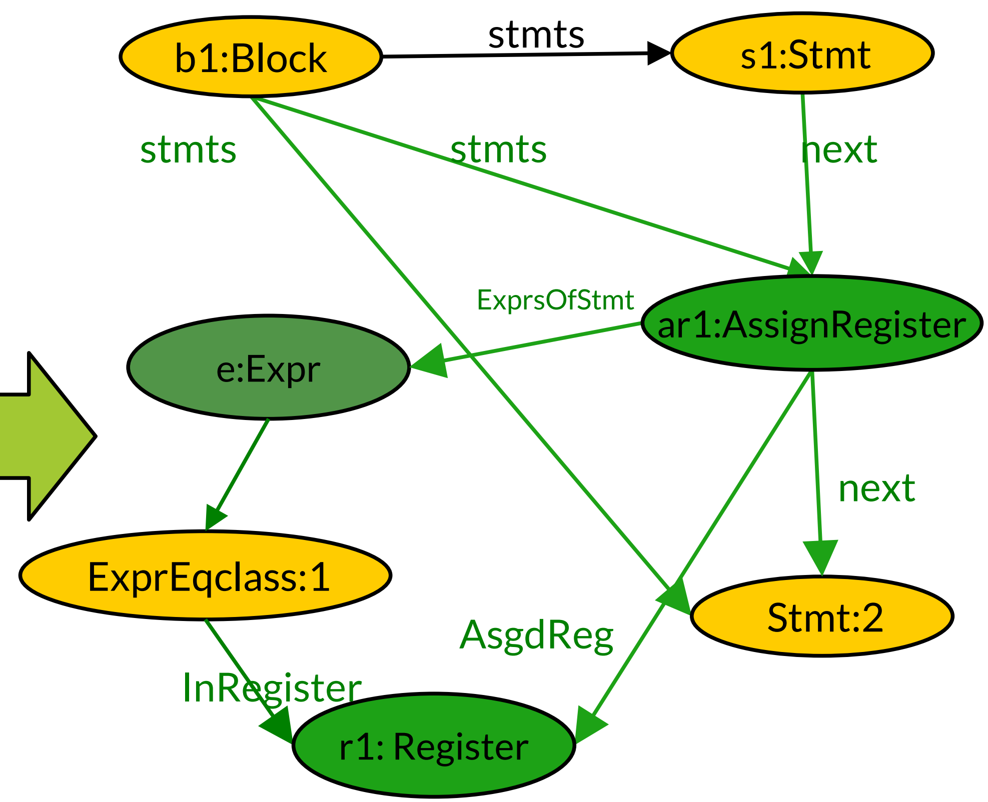
Example: Lazy Code Motion Transformation (in L2R Form)

[Aßmann00]

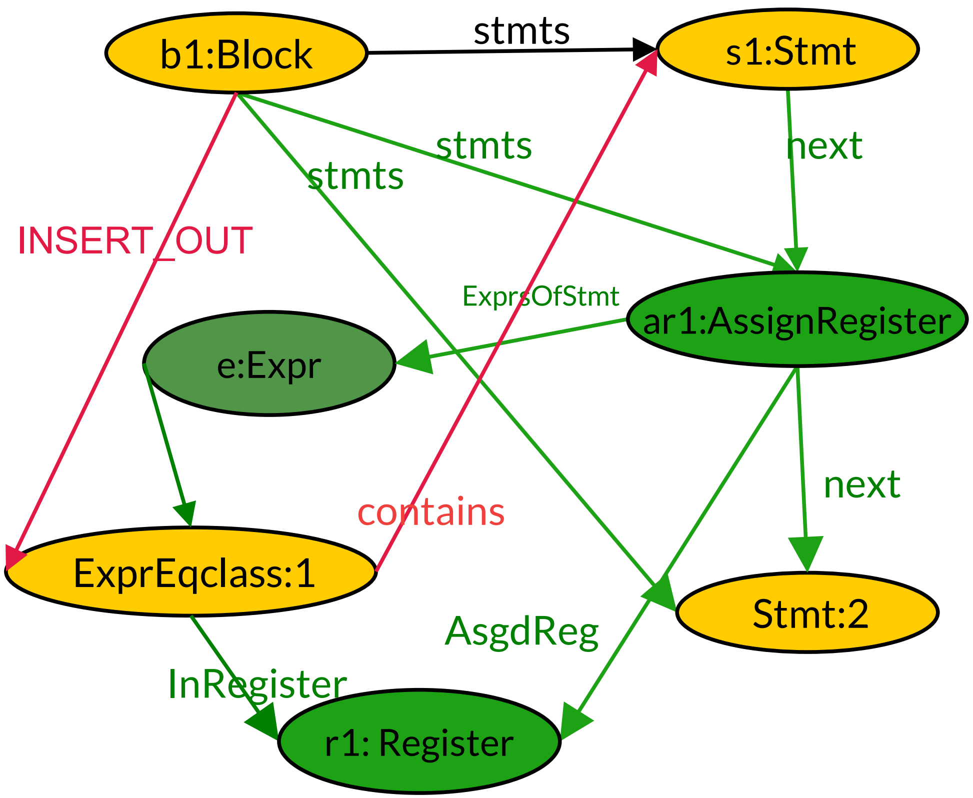
```
// GrGen
rule insertAssignRegisterAtBlockOUT (b1:Block) {
 b1 -:stmts-> s1:Stmt; s1 -n:next-> s2:Stmt;
 b1 -insert:INSERT_OUT-> ec1:ExprEqclass;
 ec1 -c:contains-> s1
 modify {
  new r1:Register; new e:Expr;    new ar:AssignRegister;
  b1 -:Stmts-> ar;    s1 -:next-> ar;   ar -:next->s2
  ec1 -:InRegister-> r1:Register;
  ar -:AsgdReg->r1,   ar -:ExprsOfStmt->s2; ar -:ExprsOfStmt->e;
  delete_edge(insert); delete_edge(c); delete_edge(n);
 }
}
```

- Insert expressions at an optimally early place and insert register assignments (ar.AssignRegister) into statement list
- INSERT_OUT indicates, at which block-exit an expression should be made available



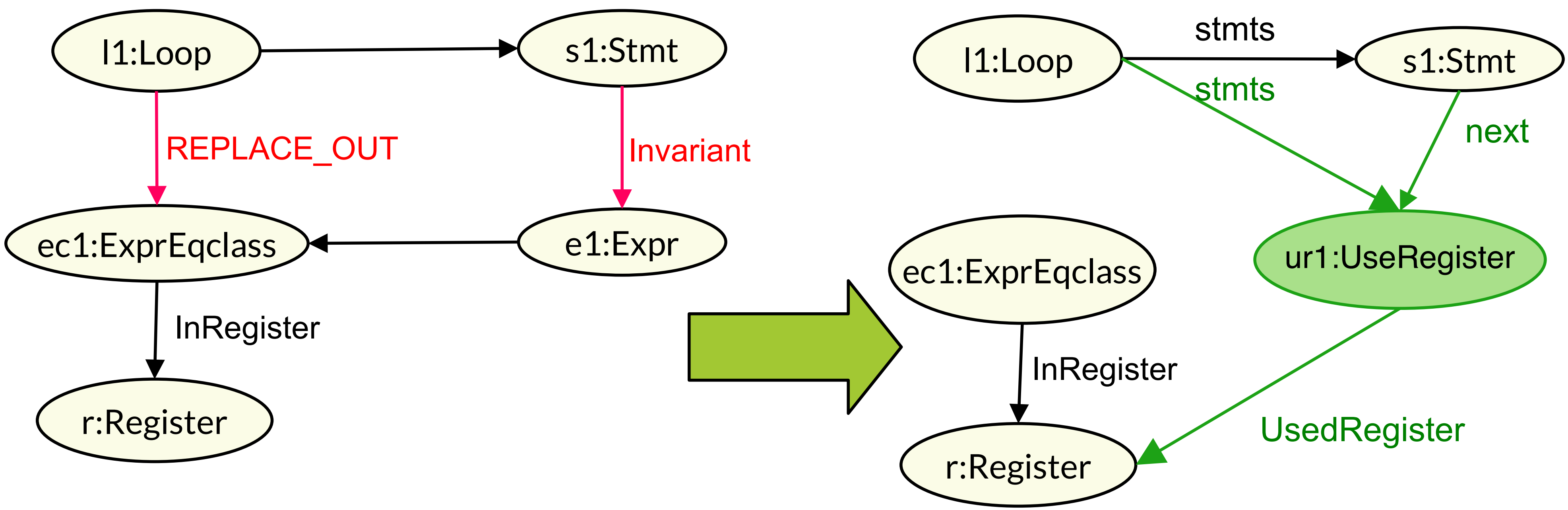


Example: Lazy Code Motion Transformation in Storyboard Notation



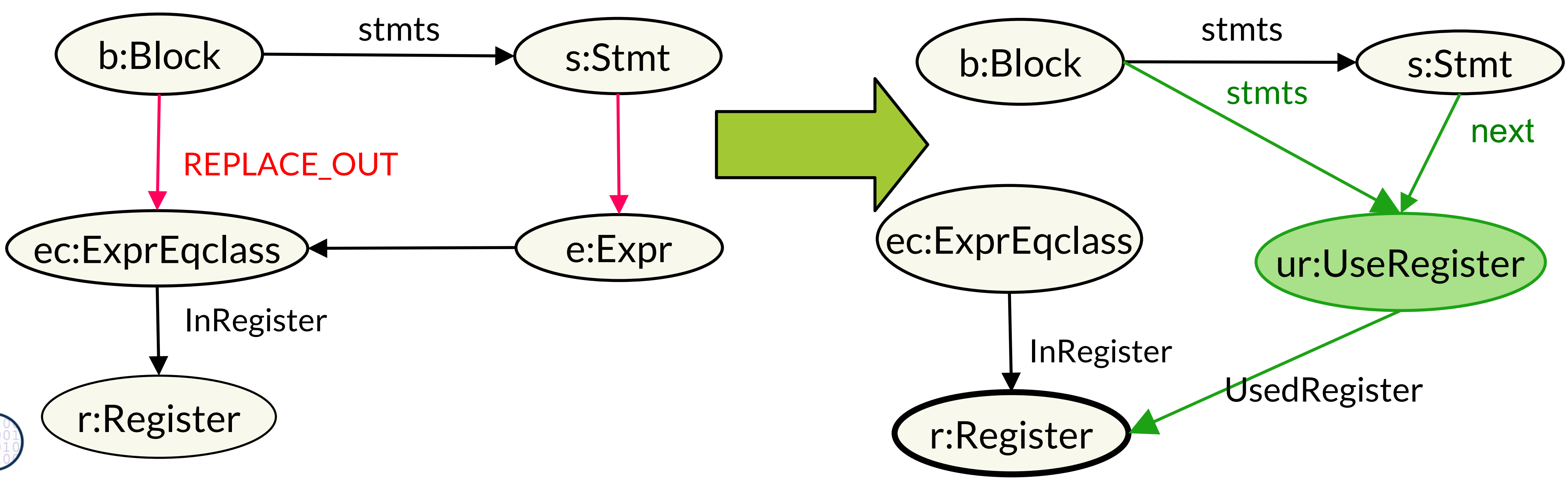
Loop Invariant Code Motion

- Loop-invariant code motion moves code before loops which is over and over computed again in the loop (loop-invariant)
- Inserts UseRegister instruction (ur1) which reuse register (r) previously stored by expression of ExprEqclass ec1



Lazy Code Motion Transformation

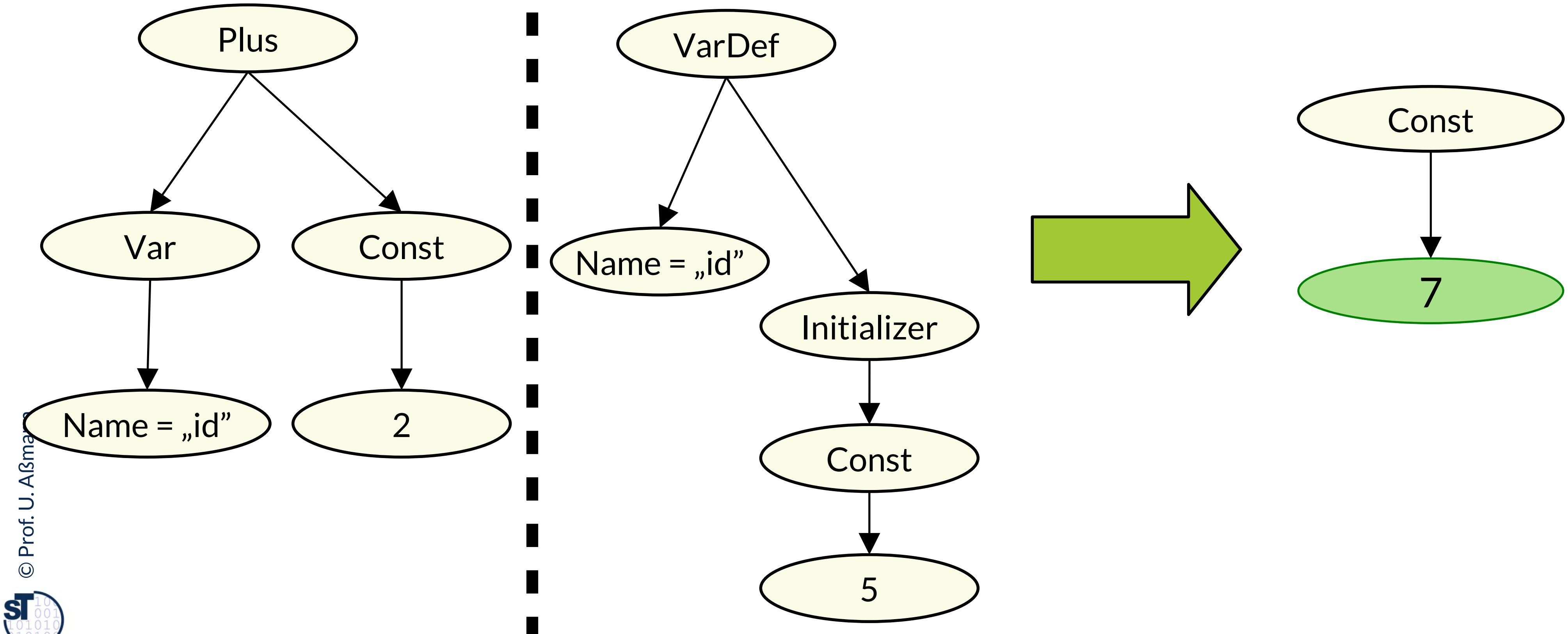
- REPLACE_OUT indicates at which block-exit an expression should no longer be computed,
- but its result should be re-used from a register



42.3. Context-Sensitive Rewritings

Extended Constant Folding as Subtractive GRS

- A term rewrite system usually works context-free, i.e., matches and rewrites only one term.
- [Lano] mostly has local rewrite rules, but context-sensitive matching is possible
- A **context-sensitive rewriting** matches a non-connected left-hand side graph with a redex.
  - Matching of one redex can be done in quadratic time, because non-connected nodes have to be pairwise compared



42.4. Program&Model Transformations with GrGen (Karlsruhe)

[GrGenManual] Edgar Jakubeit, Jakob Blomer, Rubino Geiß, The GrGen.NET User Manual Refers to GrGen.NET Release 4.4.2

- www.grgen.net
- http://www.info.uni-karlsruhe.de/software/grgen/
- Some slides courtesy to Mirko Seifert

GrGen

- GrGen: Graph Rewrite Generator from U Karlsruhe (Edgar Jakumeit)
  - Single-Pushout SPO Graph rewrite systems
- GrGen is one of the fastest graph transformation tools around
  - Proprietary DDL grgen.gm, similar to MOF (multiple inheritance, uni-, bidirectional)
  - Textual rule syntax (no graphical rule syntax)
  - Interpreter (shell)
  - Visualizer for result graphs
- Powerful language
  - Nesting of rules
  - Alternative subrules
  - Negated rules
- The following examples stem from [GrGenManual]

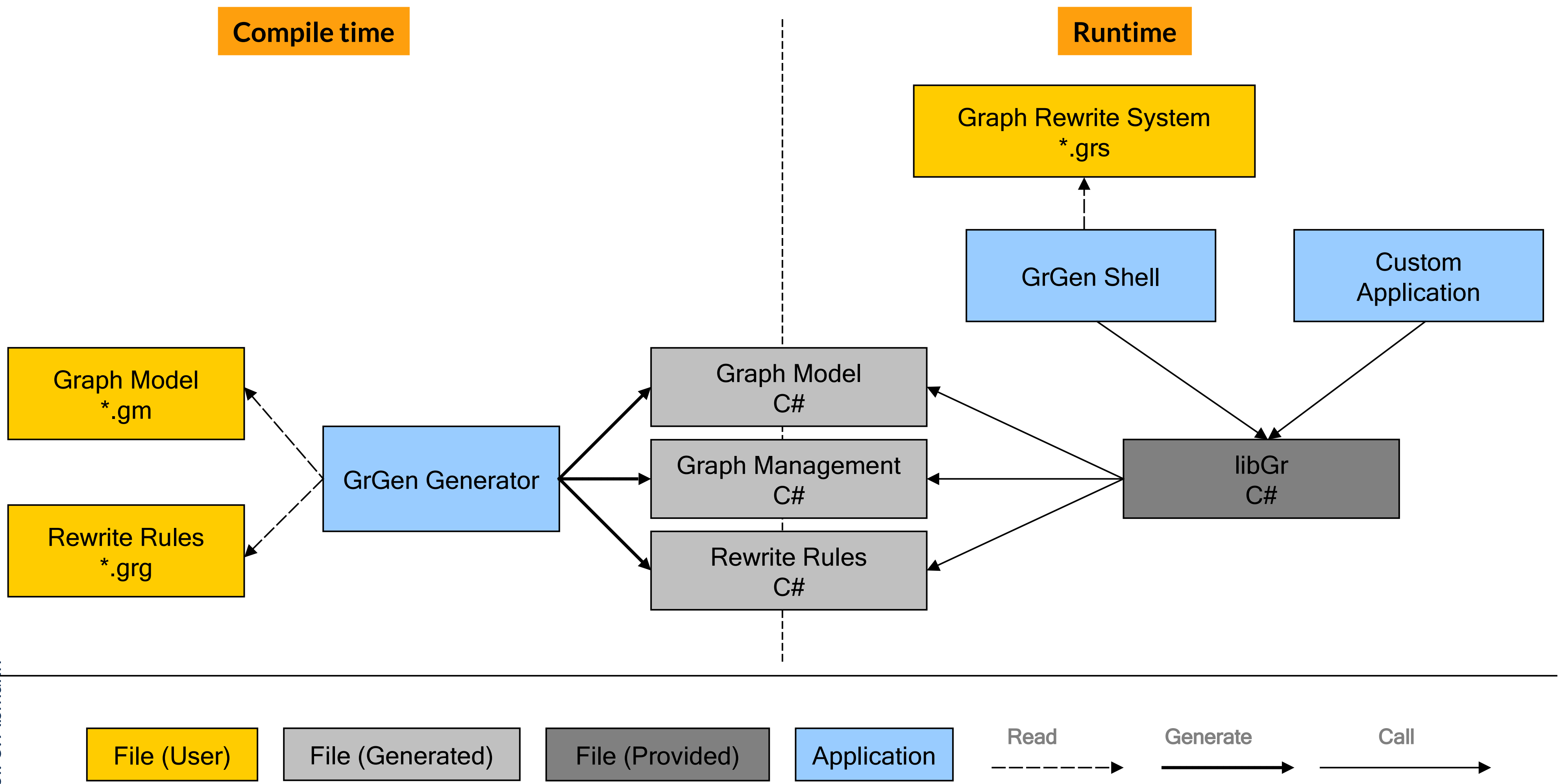
GrGen Metalanguage

- GrGen has a metalanguage / DDL similar to MOF, with
  - node, edge, and graph types with single and multiple inheritance
  - directed, typed, attributed, uni- or bidirectional (multi-)graphs
    - negative application conditions
    - alternatives
    - iterations
  - *named rules* with many flavours, can be called like procedures
- Try to specify the MOF metamodel of the last slide in GrGen DDL

What Can GrGen Do?

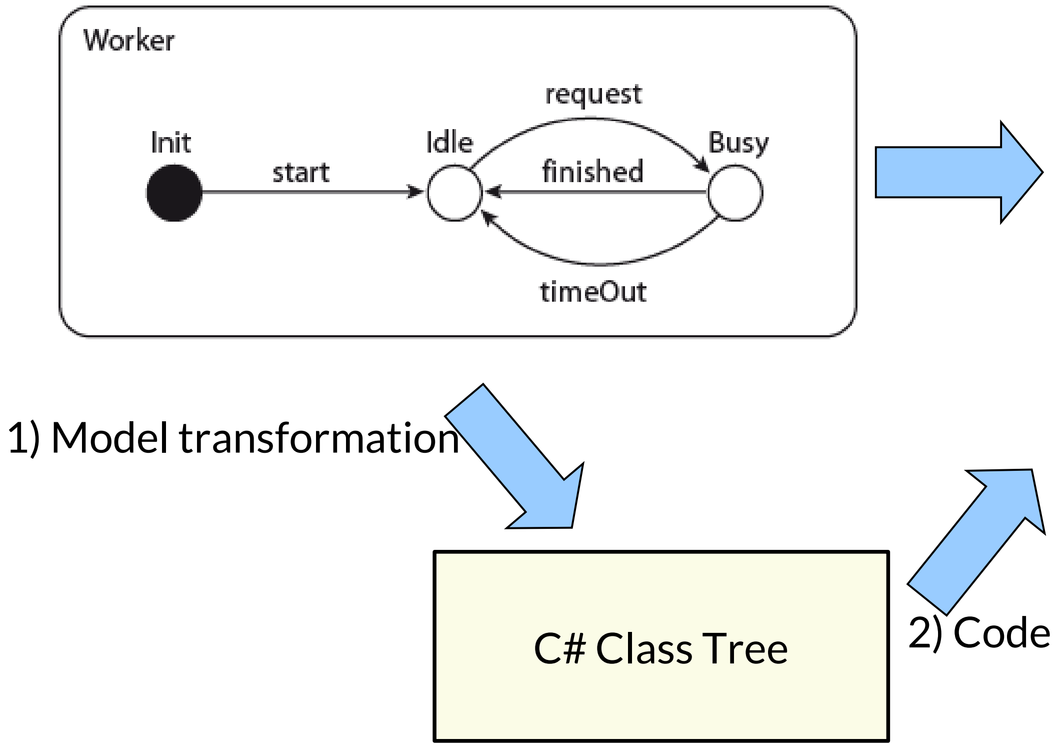
- Graph Patterns:
  - Isomorphic/Homomorphic Subgraph Matching
  - Attribute, Type Conditions (instanceof)
  - Parametrisizable
- Graph Rewrites:
  - Attribute computations
  - Casting of node types, edge types
  - Creation of nodes and edges with dynamic types
  - Modify vs. Replace mode

Build Management with GrGen



Example – Codegenerator from State Machines to C#

- Input: State Machine(s)
- Output: Simple AST for (a modifed) State Pattern implementation




```
class Worker {
    AbstractState currentState;
    Worker () {
        currentState = new InitState();
    }
    class AbstractState {
        void start() {};
        void request() {};
        void finished() {};
        void timeOut() {};
    }
    class InitState extends AbstractState {
        void start() {
            currentState = new IdleState();
        }
    }
    class IdleState extends AbstractState {
        void request() {
            currentState = new BusyState();
        }
    }
    class BusyState extends AbstractState {
        void finished() {
            currentState = new IdleState();
        }
        void timeOut() {
            currentState = new IdleState();
        }
    }
}
```

Example – Transformation of Models conforming to Metamodel of State Machine to C# Metamodel

- Input of Transformation: State Machine(s) Metamodel

  Node types: STATEMACHINE, STATE, STARTSTATE, FINALSTATE

  Edge types: TRANSITION, TRIGGER, EPSILONTRANSITION

- Output of Transformation: C# AST

  Node types: CLAZZ, INNER_CLAZZ, MEMBER, ATTRIBUTE, METHOD, CONSTRUCTOR, ASSIGNMENT

  Edge types: PARENT, LEFT, RIGHT

```
//
// Excerpt from State Machine Metamodel
//
// concrete classes of Input of Transformation
node class State { id: int;
}
// abstract classes
abstract node class SpecialState extends State;

// node inheritance
node class StartState extends SpecialState;
node class FinalState extends SpecialState;
node class StartFinalState extends StartState,
FinalState;

// concrete edge classes
edge class Transition {
   Trigger: string;
   Source, Target, Owner: State;
}
edge class Trigger extends Transition;

edge class EpsilonTransition extends Transition;
```

Example – Syntax-Directed Rules Required for Model Transformation

1. sm2class – translate state machine model to class model
2. owner2parent
3. removeStatemachine

// transformations for transforming single snippets:

4. addStateAttribute
5. addAbstractStateClass
6. addMethodsForEvent
7. addMethodForTransition
8. addConstructor
9. state2class – translate state model to class model

// transformations for transforming single snippets:

10. moveMemberToClass
11. moveAssignmentToClass
12. moveStateClass
13. removeState

1) // checking Wellformedness conditions

1. checkStartState
2. checkDoublettes

Example - Test Rules Check Conditions in the Graph


```
#using „stateMachine.gm“

test checkStartState { x:StartState;

        negative { x;

        y:StartState; }

}

test checkDoublettes { negative {

     x:State -e:Transition-> y:State;

     hom(x,y);

     x -doublette:Transition-> y;

     if {typeof(doublette) == typeof(e);}

     if { ((typeof(e) == EpsilonTransition)
        || (e.Trigger == doublette.Trigger)); }

    }

}
```

Example - Test Rules Check Conditions in the Graph


```
#using „stateMachine.gm“

rule forwardTransition {
  x:State -:EpsilonTransition-> y:State -e:Transition-> z:State; hom(x,y,z);
  negative {
    x -exists:Transition-> z;
    if {typeof(exists) == typeof(e);}
    if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
   }
   modify {
    x -forward:typeof(e)-> z;
    eval {forward.Trigger = e.Trigger;}
   }
}
```

```
#using „stateMachine.gm“

rule addStartFinalState {
  x:StartState -:EpsilonTransition-> :FinalState; modify {
  y:StartFinalState<x>; ---
  emit("Start␣state␣(", x.id, ")␣mutated␣into␣a␣start-and-final␣state"); }
}
rule addFinalState {
  x:State -:EpsilonTransition-> :FinalState; if {typeof(x) < SpecialState;}
  modify { y:FinalState<x>; }
}
rule removeEpsilonTransition { -:EpsilonTransition->; replace {} }
```

GrGen Shell Language (Runtime)

- GrGen has an interactive shell in which graphs can be
  - allocated
  - layouted for print (`show graph`)
  - rewritten by graph rewriting procedures

```
// creating a state machine in GrGenShell
new graph removeEpsilons "StateMachineGraph"
new :StartState($=S, id=0)
new :FinalState($=F, id=3)
new :State($="1", id=1)
new :State($="2", id=2)
new @(S)-:Transition(Trigger="a")-> @("1") new @("1")-:Transition(Trigger="b")->
@("2") new @("2")-:Transition(Trigger="c")-> @(F) new @(S)-:EpsilonTransition-> @("2")
new @("1")-:EpsilonTransition-> @(F) new @(S)-:EpsilonTransition-> @(F)
show graph ycomp
```

Example 2 - Definition of Context-oriented Petri Nets in GrGen Shell Graph Definition Language (DDL)


```
// Context-oriented Petri Nets
node class ContextNet extends NIdent { }
edge class contexts
    connect ContextNet[+] --> Context[1];
node class Context extends NIdent {
    active: int = 0;
    bound: int = 1;
}
// weak inclusion relation of contexts: on activation/deactivation trigger this with target
// act(source) -> act(target);  deact(source) -> deact(target)
// act(target) ->; deact(target) ->
directed edge class weak_inclusion  connect Context[*] --> Context[*];

// exclusion: both contexts cannot be active at same time
// act(source) -> deact(target)
// act(target) -> deact(source)
undirected edge class exclusion      connect Context[*] -- Context[*];

directed edge class composition      connect Context[*] --> Context[*];
```

Example 2 - Definition of Context-oriented Petri Nets in GrGen Shell Graph Definition Language (DDL) (ctd)


```
// strong inclusion : when target gets deactivated, the source also
// empty triangle
// act(source) -> act(target)
// deact(source) -> deact(target)
// act(target) ->
// deact(target) -> deact(source) -> deact(target)
directed edge class strong_inclusion // full triangle
    connect Context[*] --> Context[*];

// requirement: context can only be activated when target is already
// empty triangle
// act(source) -> only if already: act(target)
// deact(source) ->
// act(target) ->
// deact(target) -> deact(source)
directed edge class requirement // inverse full triangle
    connect Context[*] --> Context[*];
```

GrGen Shell Language for Allocating Graphs


```
new cign:Context(id="Ignore")

new clow:Context(id="LowBattery")

new chig:Context(id="HighBattery")

new cvid:Context(id="VideoCall")

new cdon:Context(id="DoNotDisturb")

new cred:Context(id="Redirect")

new cemu:Context(id="Emergency+Unavailable")

new cfro:Context(id="FrontCamera")

new cuna:Context(id="Unavailable")

# edges

new cign-:exclusion-cuna

new cign-:exclusion-cred

new cuna-:exclusion-cred

new clow-:exclusion-chig


new clow-:weak_inclusion->cign

new cred-:weak_inclusion->cdon

new cvid-:strong_inclusion->cfrz
```

GrGen Shell Language: Piping of Graph Rewriting Phases


```
exec p2c_Init(1)
exec [transform_1()]
exec [transform_2()]
exec transform_3_add_logic_or()*
exec CircCleanup()


debug exec (net:Page) =
InitContext2Petri() | [transform()] | ComposePage(net)
| composition_invariant_new()* | composition_invariant_new2() | Cleanup()
```

Normal Rules with Nested Alternatives

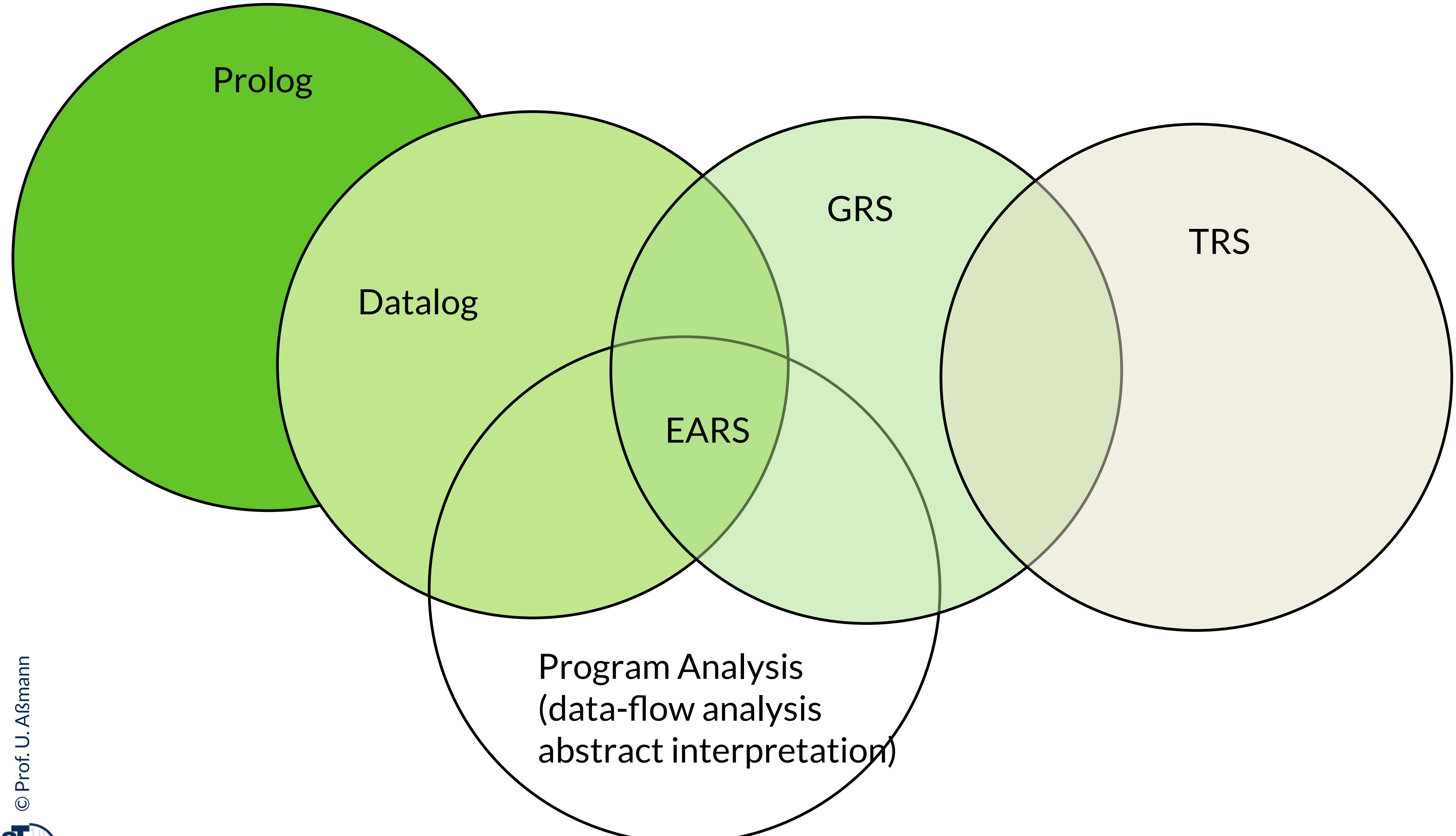

```
rule foldCond {
        cond:Cond -df0:Dataflow-> c0:Const;
        falseBlock:Block -falseEdge:False-> cond;
        trueBlock:Block -trueEdge:True-> cond;
  alternative {
      TrueCond {
        if { c0.value == 1; } modify {
                delete(falseEdge);
                -jmpEdge:Controlflow<trueEdge>->; }
      }
      FalseCond {
        if { c0.value == 0; } modify {
                delete(trueEdge);
                -jmpEdge:Controlflow<falseEdge>->; }
        }
      }
   }
   modify { delete(df0); jmp:Jmp<cond>; }
}
```

42.5 More on the Logic-Graph Isomorphism

Results

- Theory: Graph rewriting, DATALOG and data-flow analysis have a common core:
  - EARS build the bridge between graph analysis and transformations
- Exhaustive GRS: If a termination graph can be identified, a GRS terminates.
- Program optimization:
  - Spezification of program optimizations is possible with graph rewrite systems. Short specifications, fewer effort.
  - Practically usable optimizer components can be generated.
- Uniform Specification of Analysis and Transformation
  - If the program analysis (including abstract interpretation) is specified with GRS, it can be unified with program transformation
- Limitations
  - Several optimizations can be specified with GRS which are not exhaustive (peephole optimization, constant propagation with partial evaluation).
  - As general rule embedding is not allowed, a rule only matches a fixed number of nodes.
  - Thus those transformations, which refer to an arbitrary set of nodes, cannot be specified.

The Common Core of Logic, Rewriting and Program Analysis



An Old Citation

There clearly remains more work to be done in the following areas:

- **discovery of other properties of transformations that appear to have relevance to code optimization,**
- **development of simple tests of these properties, and**
- **the use of these properties to construct efficient and effective optimization algorithms that apply the transformations involved.**

Aho, Sethi, Ullmann in Code Optimization and Finite Church-Rosser Systems, 1972

The End

- Explain a connected graph pattern for local graph rewriting. What is a disconnected graph pattern for context-sensitive graph rewriting?
- What does it mean when GRS are exhaustive (XGRS)?

- Many GrGen examples are from Carl Mai
  - https://petrinets.pages.st.inf.tu-dresden.de/adaptive-petrinets/index.html
  - https://git-st.inf.tu-dresden.de/adaptive_petrinets/reconfnet

42.4. Model Transformations with ATL

ATLAS Transformation Language (ATL)
http://www.eclipse.org/atl/

Tools for Model-Driven Software Development

- In MDSD and MDA, horizontal and vertical model transformations should be specified with graph rewrite systems
- Example tools:
  - JastAdd RAGs (Java)
  - GrGen (C#)
  - **ATL** in Eclipse EMOF

ATL Integrates OCL as Query Language


```
// Transitive Closure  in ATL, with a recursive OCL query
rule computeTransitiveClosureBaseCase {
  from node: Node (
    // possible to call OCL expressions
    node->baserelation.collect( e | e.baserelation)->flatten() );
  )
  to newNode mapsTo node (
    // set new transitive relation
    newNode->transitiverelation <- node->baserelation
  )
}
rule computeTransitiveClosureRecursiveCase {
  from node: Node (
    node->transitiverelation.collect( e | e.baserelation)->flatten() );
  )
  to newNode mapsTo node (
    // set new transitive relation
    newNode->transitiverelation <- node->transitiverelation
  )
}
```

Terminology for Automated Graph Rewriting

- **Graph rewrite rule:** rule (left, right hand side) to match left-hand side in the graph and to transform it to the right-hand side
- **Graph rewrite system:** set of graph rewrite rules
- **Start graph (axiom):** input graph to rewriting
- **Graph rewrite problem:** a graph rewrite system applied to a start graph
- **Manipulated graph (host graph):** graph which is rewritten in graph rewrite problem
- **Redex**: (reducible expression) application place of a rule in the manipulated graph
- **Derivation:** a sequence of rewrite steps on the manipulated graph, starting from the start graph and ending in the normal form
- **Normal form:** result graph of rewriting; manipulated graphs without further redex
- **Unique normal form:** unique result of a rewrite system, applied to one start graph
- **Terminating GRS:** rewrite system that stops after finite number of rewrites
- **Confluent GRS:** two derivations always can be commuted, resp. joined together to one result
- **Convergent GRS:** rewrite system that always yields unique results (terminating and confluent)

42. GraphWare: Languages for Graph Transformations and Rewriting

Refactoring, Improvement of Large Models

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 21-0.2, 29.01.22

1) Graph rewriting
2) Complex local graph rewritings
3) Context-sensitive graph rewritings
4) GrGen
5) More on the Graph-Logic Isomorphism

Obligatory Literature

- Kevin Lano. Catalogue of Model Transformations
  - http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf
- 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
- Jakumeit, E., Buchwald, S. & Kroll, M. GrGen.NET. Int J Softw Tools Technol Transfer 12, 263–271 (2010). https://doi.org/10.1007/s10009-010-0148-8
- [GrGenManual] e. Jakumeit, J. Blomer, R. Geiß. The GrGen.NET User Manual Refers to GrGen.NET Release 6.1.1.
  - www.grgen.net
- Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/

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

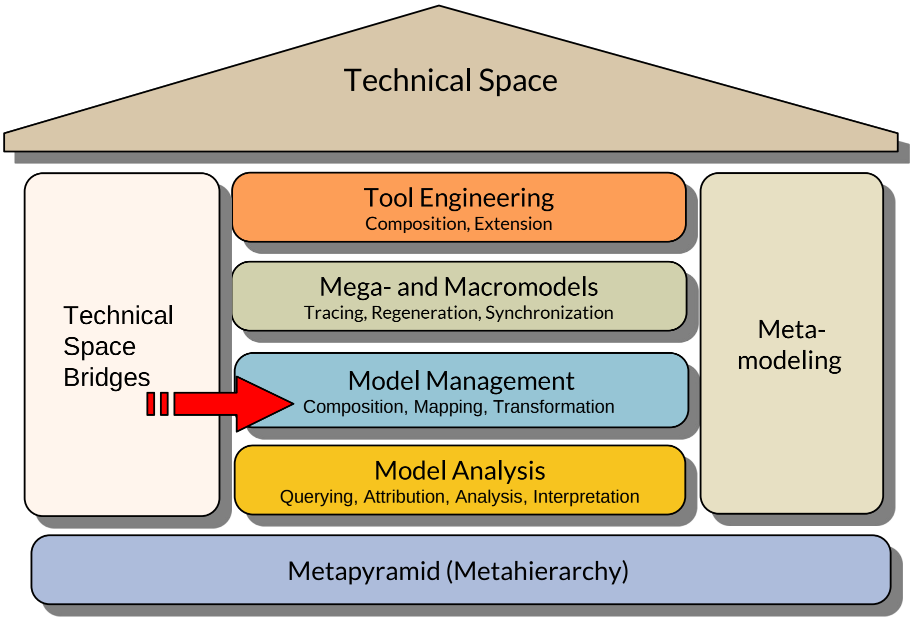
Other Literature

- Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006), pages 1188—1195.
  - http://atlanmod.emn.fr/bibliography/SAC06a
- Tutorial über ATL "Families2Persones"
  - http://www.eclipse.org/m2m/atl/doc/ATLUseCase_Families2Persons.ppt
- ATL Zoo von Beispielen
  - http://www.eclipse.org/m2m/atl/atlTransformations
- A Comparison of ATL and Story-Driven Modeling (Fujaba-style GRS)

http://www.es.tu-darmstadt.de/fileadmin/download/publications/spatzina/PP_AGTIVE_2011.pdf

- Implementation in ATL
  - http://www.eclipse.org/m2m/atl/atlTransformations/EquivalenceAttributesAssociations/EquivalenceAttributesAssociations.pdf

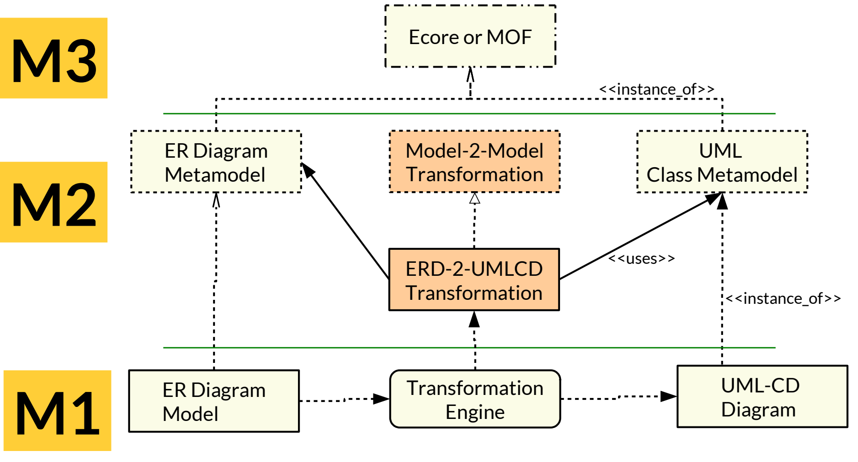
Q10: The House of a Technical Space



42.1. Graph Rewriting for Code and Model Creation, Transformations, Translation

Model Transformations and Their Metamodels

- Model transformations defined in Layer M_{i+1} specify how to transform models on M_i
  - Source and target metamodel are connected by a metamodel of a rule-based transformation language
- **Benefit**: Transformation can be reused for all models, which are instances of the source meta-model



Needs of Graph Rewriting

- M3: Graph-enabling Metalanguage (Ecore, MOF, ERD, RDFS, OWL)
- M2: Metamodel for Nodes and Edges (DDL)
  - MOF, EMOF, tool-specific, etc.
- M2: Metamodel for Rule Language
  - LR Form (left to right rule)
  - Fujaba Storyboard Notation
  - GrGen metamodel
- M1: Node and edge allocators (factories)
  - Graph libraries such as Jgrapht.org
- M0: graph pools with nodes and edges
  - To limit rewriting

Applications of Graph Rewrite Systems for *Transformations* (Graphersetzungssysteme)

- **Concrete and abstract Interpretation** of code and models [Rensink]
- MDSD tools need model transformations
  - Model transformations (Alexander Christoph)
  - Model aspect weaving (Aßmann, Heidenreich, many others)
  - Creation of more specific models in MDA, including the computation of trace create links [Taentzer]
  - Refinement in design [Lano, Schürr, Lewerentz]
  - Refactoring [Mens]
- Compilation and Translation of code and models [Nagl, Aßmann]
- Analysis [Tip, Reps]
  - Slicing
  - Interprocedural analysis with graph reachability
- Optimization [Aßmann]
  - Global code transformations, such as lazy and busy code motion (loop invariant code motion)
- Configuration management [Westfechtel]

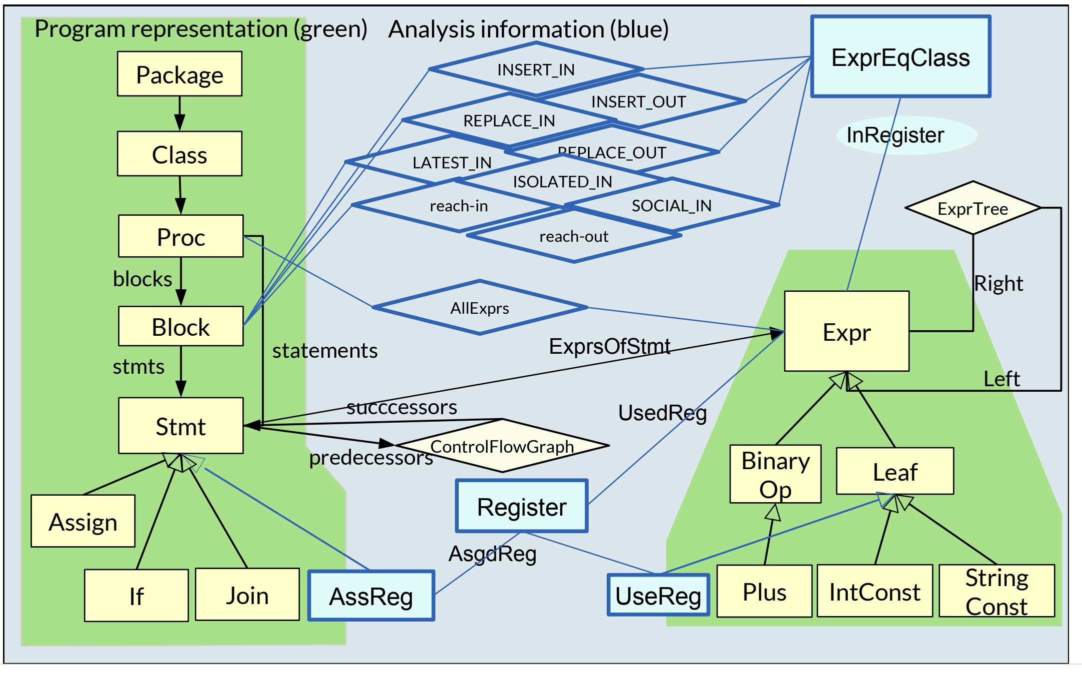
Model Transformation and Optimization with Graph Rewriting

- Use the **graph-logic-isomorphism** [Courcelle]: 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 addition rewrite systems (EARS) to query, analyze, map the graphs to each other
- Use graph rewrite systems (GRS) to create, construct, and augment the graphs
  - Transform the graphs
  - Generate code
- Preferably, the GRS should terminate (XGRS, exhaustive GRS)

Specification Process For Transformation with Deep Analysis

1) Specification of the data model (graph schema)
   - Specification of the 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 infered predicates over the program objects) as objects or relationships
2) „Flat“ program analysis (preparing the abstract interpretation)
   - Querying graphs, enlarging graphs
   - Materializing implicit knowledge to explicit knowledge
3) „Deep Analysis“: 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) Program transformation (optimization)
   - Transforming the program representation

Q14: A Simple Program Code&Model Schema in MOF



42.2. Complex Local Graph Rewritings

- On Dags (with joins) and Graphs (with cycles)

Code Optimizations Expressible by Local Graph Rewritings

- Local transformations of the program representation
  - copy propagation (copy of expression is loaded to register and reused)
  - constant propagation (constants instead of expressions)
  - branch optimization
  - loop optimizations (unrolling etc.)
  - strength reduction (Multiplications to additions)
  - idiom recognition (pattern matching of complex patterns)
  - dead code elimination (elimination of non-reachable code)

Model Transformations Expressible by Graph Rewritings

[Lano]

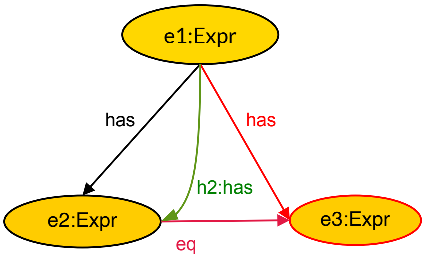
- Transformations of the inheritance hierarchy:
  - Flattening (Reachability)
  - Removal of redundant inheritances
  - Refactorings
    - Split classes
    - Merge classes
    - Move class
- Support during refinement
  - Flattening aggregation, composition, multiple inheritance
- Optimizations:
  - Peephole optimizations (local transformations)
- Generation of dependent models
  - Export to exchange file formats, such as JSON and XML

Example: Peephole Transformation “Local Sharing of Equivalent Subexpressions”


```
if has(Expr1, Expr2),
   has(Expr1, Expr3),
   eq(Expr2,Expr3)
then
   delete Expr3;
   h2:has(Expr1,Expr3)
;
```

- Share common subexpressions
- Here e2

```
// GrGen
rule foldCommonSubexpression(e1:Expr) {
   e1 -:has-> e2:Expr;
   e1 -h1:has-> e3:Expr;
   e2 -e:eq->e3;
   modify { e1:-h2:has-> e2 ; delete(e3);
   delete_edge(h1); delete_edge (e);
 }
}
```

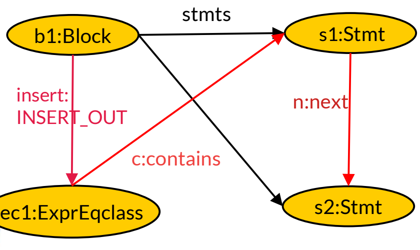


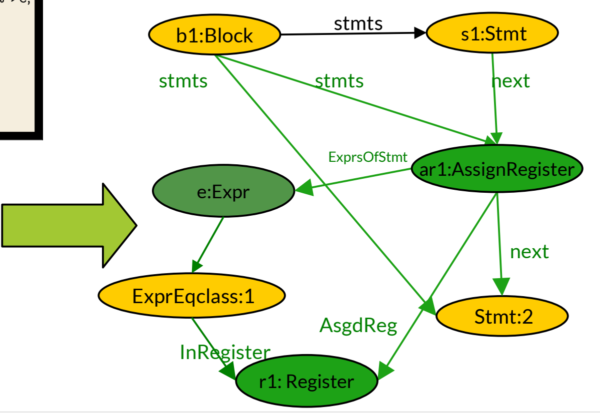
Example: Lazy Code Motion Transformation (in L2R Form)

[Aßmann00]

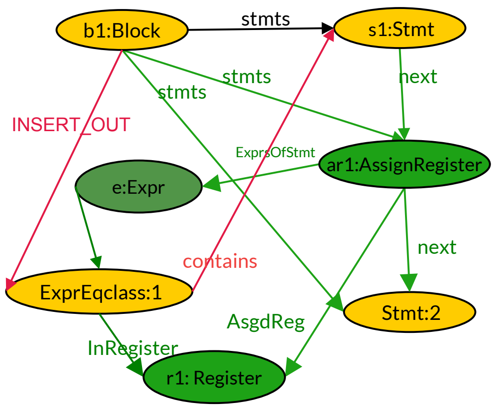
```
// GrGen
rule insertAssignRegisterAtBlockOUT (b1:Block) {
  b1 -:stmts-> s1:Stmt; s1 -n:next-> s2:Stmt;
  b1 -insert:INSERT_OUT-> ec1:ExprEqclass;
  ec1 -c:contains-> s1
  modify {
   new r1:Register; new e:Expr;    new ar:AssignRegister;
   b1 -:Stmts-> ar;    s1 -:next-> ar;   ar -:next->s2
   ec1 -:InRegister-> r1:Register;
   ar -:AsgdReg->r1,  ar -:ExprsOfStmt->s2; ar -:ExprsOfStmt->e;
   delete_edge(insert); delete_edge(c); delete_edge(n);
  }
}
```

- Insert expressions at an optimally early place and insert register assignments (ar.AssignRegister) into statement list
- INSERT_OUT indicates, at which block-exit an expression should be made available



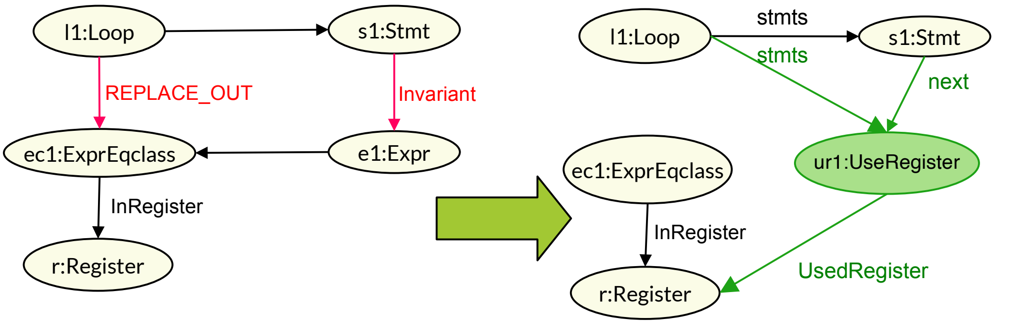


Example: Lazy Code Motion Transformation in Storyboard Notation



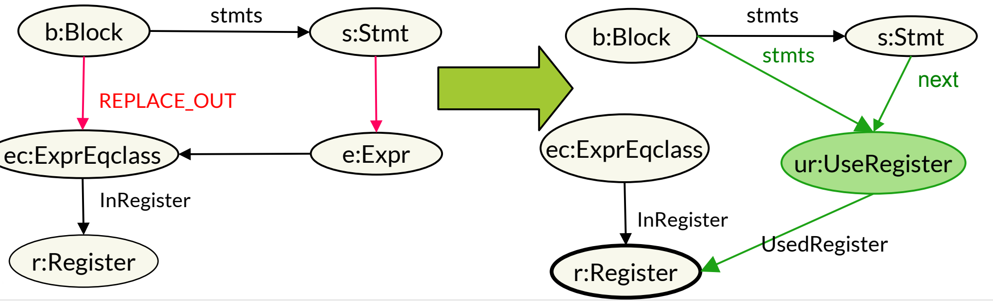
Loop Invariant Code Motion

- Loop-invariant code motion moves code before loops which is over and over computed again in the loop (loop-invariant)
- Inserts UseRegister instruction (ur1) which reuse register (r) previously stored by expression of ExprEqclass ec1



Lazy Code Motion Transformation

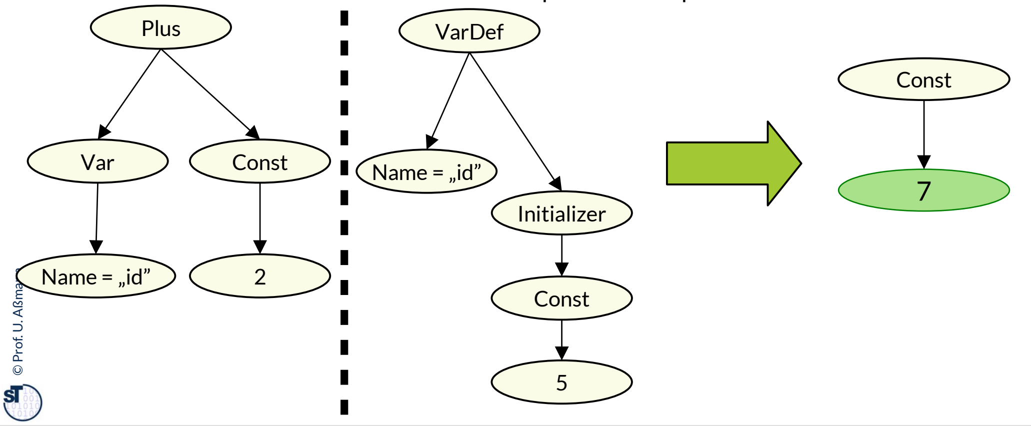
- REPLACE_OUT indicates at which block-exit an expression should no longer be computed,
- but its result should be re-used from a register



42.3. Context-Sensitive Rewritings

Extended Constant Folding as Subtractive GRS

- A term rewrite system usually works context-free, i.e., matches and rewrites only one term.
- [Lano] mostly has local rewrite rules, but context-sensitive matching is possible
- A **context-sensitive rewriting** matches a non-connected left-hand side graph with a redex.
  - Matching of one redex can be done in quadratic time, because non-connected nodes have to be pairwise compared



42.4. Program&Model Transformations with GrGen (Karlsruhe)

[GrGenManual] Edgar Jakubeit, Jakob Blomer, Rubino Geiß, The GrGen.NET User Manual Refers to GrGen.NET Release 4.4.2

- www.grgen.net
- http://www.info.uni-karlsruhe.de/software/grgen/
- Some slides courtesy to Mirko Seifert

GrGen

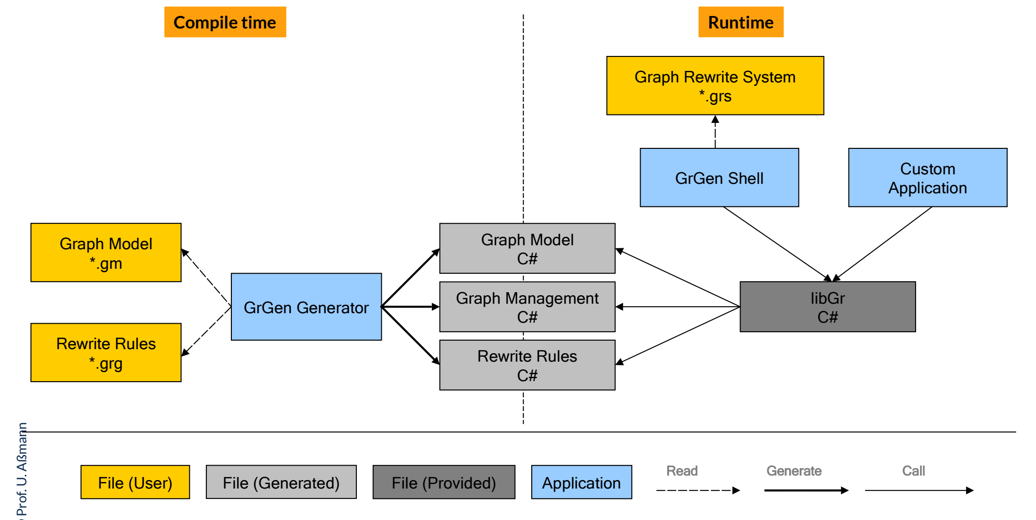
- GrGen: Graph Rewrite Generator from U Karlsruhe (Edgar Jakumeit)
  - Single-Pushout SPO Graph rewrite systems
- GrGen is one of the fastest graph transformation tools around
  - Proprietary DDL grgen.gm, similar to MOF (multiple inheritance, uni-, bidirectional)
  - Textual rule syntax (no graphical rule syntax)
  - Interpreter (shell)
  - Visualizer for result graphs
- Powerful language
  - Nesting of rules
  - Alternative subrules
  - Negated rules
- The following examples stem from [GrGenManual]

GrGen Metalanguage

- GrGen has a metalanguage / DDL similar to MOF, with
  - node, edge, and graph types with single and multiple inheritance
  - directed, typed, attributed, uni- or bidirectional (multi-)graphs
    - negative application conditions
    - alternatives
    - iterations
  - *named rules* with many flavours, can be called like procedures
- Try to specify the MOF metamodel of the last slide in GrGen DDL

What Can GrGen Do?

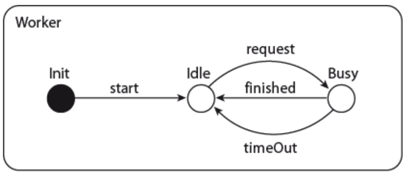
- Graph Patterns:
  - Isomorphic/Homomorphic Subgraph Matching
  - Attribute, Type Conditions (instanceof)
  - Parametrisizable
- Graph Rewrites:
  - Attribute computations
  - Casting of node types, edge types
  - Creation of nodes and edges with dynamic types
  - Modify vs. Replace mode



- Graph model describes the node and edge types
- Rewrite Rules contains the graph pattern matching and transformation rules

Example – Codegenerator from State Machines to C#

- Input: State Machine(s)
- Output: Simple AST for (a modifed) State Pattern implementation



1) Model transformation

C# Class Tree

2) Code generation

```
class Worker {
    AbstractState currentState;
    Worker () {
        currentState = new InitState();
    }
    class AbstractState {
        void start() {};
        void request() {};
        void finished() {};
        void timeOut() {};
    }
    class InitState extends AbstractState {
        void start() {
            currentState = new IdleState();
        }
    }
    class IdleState extends AbstractState {
        void request() {
            currentState = new BusyState();
        }
    }
    class BusyState extends AbstractState {
        void finished() {
            currentState = new IdleState();
        }
        void timeOut() {
            currentState = new IdleState();
        }
    }
}
```

Example – Transformation of Models conforming to Metamodel of State Machine to C# Metamodel

- Input of Transformation: State Machine(s) Metamodel

  Node types: STATEMACHINE, STATE, STARTSTATE, FINALSTATE

  Edge types: TRANSITION, TRIGGER, EPSILONTRANSITION

- Output of Transformation: C# AST

  Node types: CLAZZ, INNER_CLAZZ, MEMBER, ATTRIBUTE, METHOD, CONSTRUCTOR, ASSIGNMENT

  Edge types: PARENT, LEFT, RIGHT

```
//
// Excerpt from State Machine Metamodel
//
// concrete classes of Input of Transformation
node class State { id: int;
}
// abstract classes
abstract node class SpecialState extends State;

// node inheritance
node class StartState extends SpecialState;
node class FinalState extends SpecialState;
node class StartFinalState extends StartState,
FinalState;

// concrete edge classes
edge class Transition {
   Trigger: string;
   Source, Target, Owner: State;
}
edge class Trigger extends Transition;

edge class EpsilonTransition extends Transition;
```

Example – Syntax-Directed Rules Required for Model Transformation

1. sm2class – translate state machine model to class model
2. owner2parent
3. removeStatemachine

// transformations for transforming single snippets:

4. addStateAttribute
5. addAbstractStateClass
6. addMethodsForEvent
7. addMethodForTransition
8. addConstructor
9. state2class – translate state model to class model

// transformations for transforming single snippets:

10. moveMemberToClass
11. moveAssignmentToClass
12. moveStateClass
13. removeState

1) // checking Wellformedness conditions

1. checkStartState
2. checkDoublettes

Example - Test Rules Check Conditions in the Graph


```
#using „stateMachine.gm“
test checkStartState { x:StartState;
        negative { x;
        y:StartState; }
}
test checkDoublettes { negative {
     x:State -e:Transition-> y:State;
     hom(x,y);
     x -doublette:Transition-> y;
     if {typeof(doublette) == typeof(e);}
     if { ((typeof(e) == EpsilonTransition)
        || (e.Trigger == doublette.Trigger)); }
    }
}
```

Example - Test Rules Check Conditions in the Graph


```
#using „stateMachine.gm“
rule forwardTransition {
   x:State -:EpsilonTransition-> y:State -e:Transition-> z:State; hom(x,y,z);
   negative {
     x -exists:Transition-> z;
     if {typeof(exists) == typeof(e);}
     if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
    }
    modify {
      x -forward:typeof(e)-> z;
      eval {forward.Trigger = e.Trigger;}
    }
}
```

```
#using „stateMachine.gm“
rule addStartFinalState {
  x:StartState -:EpsilonTransition-> :FinalState; modify {
  y:StartFinalState<x>; ---
  emit("Start␣state␣(", x.id, ")␣mutated␣into␣a␣start-and-final␣state"); }
}
rule addFinalState {
  x:State -:EpsilonTransition-> :FinalState; if {typeof(x) < SpecialState;}
  modify { y:FinalState<x>; }
}
rule removeEpsilonTransition { -:EpsilonTransition->; replace {} }
```

GrGen Shell Language (Runtime)

- GrGen has an interactive shell in which graphs can be
  - allocated
  - layouted for print (`show graph`)
  - rewritten by graph rewriting procedures

```
// creating a state machine in GrGenShell
new graph removeEpsilons "StateMachineGraph"
new :StartState($=S, id=0)
new :FinalState($=F, id=3)
new :State($="1", id=1)
new :State($="2", id=2)
new @(S)-:Transition(Trigger="a")-> @("1") new @("1")-:Transition(Trigger="b")->
@("2") new @("2")-:Transition(Trigger="c")-> @(F) new @(S)-:EpsilonTransition-> @("2")
new @("1")-:EpsilonTransition-> @(F) new @(S)-:EpsilonTransition-> @(F)
show graph ycomp
```

Example 2 - Definition of Context-oriented Petri Nets in GrGen Shell Graph Definition Language (DDL)


```
// Context-oriented Petri Nets
node class ContextNet extends NIdent { }
edge class contexts
    connect ContextNet[+] --> Context[1];
node class Context extends NIdent {
    active: int = 0;
    bound: int = 1;
}
// weak inclusion relation of contexts: on activation/deactivation trigger this with target
// act(source) -> act(target);  deact(source) -> deact(target)
// act(target) ->; deact(target) ->
directed edge class weak_inclusion  connect Context[*] --> Context[*];

// exclusion: both contexts cannot be active at same time
// act(source) -> deact(target)
// act(target) -> deact(source)
undirected edge class exclusion     connect Context[*] -- Context[*];

directed edge class composition     connect Context[*] --> Context[*];
```

Example 2 - Definition of Context-oriented Petri Nets in GrGen Shell Graph Definition Language (DDL) (ctd)


```
// strong inclusion : when target gets deactivated, the source also
// empty triangle
// act(source) -> act(target)
// deact(source) -> deact(target)
// act(target) ->
// deact(target) -> deact(source) -> deact(target)
directed edge class strong_inclusion // full triangle
    connect Context[*] --> Context[*];

// requirement: context can only be activated when target is already
// empty triangle
// act(source) -> only if already: act(target)
// deact(source) ->
// act(target) ->
// deact(target) -> deact(source)
directed edge class requirement // inverse full triangle
    connect Context[*] --> Context[*];
```

GrGen Shell Language for Allocating Graphs


```
new cign:Context(id="Ignore")
new clow:Context(id="LowBattery")
new chig:Context(id="HighBattery")
new cvid:Context(id="VideoCall")
new cdon:Context(id="DoNotDisturb")
new cred:Context(id="Redirect")
new cemu:Context(id="Emergency+Unavailable")
new cfro:Context(id="FrontCamera")
new cuna:Context(id="Unavailable")
# edges
new cign-:exclusion-cuna
new cign-:exclusion-cred
new cuna-:exclusion-cred
new clow-:exclusion-chig

new clow-:weak_inclusion->cign
new cred-:weak_inclusion->cdon
new cvid-:strong_inclusion->cfrz
```

GrGen Shell Language: Piping of Graph Rewriting Phases


```
exec p2c_Init(1)
exec [transform_1()]
exec [transform_2()]
exec transform_3_add_logic_or()*
exec CircCleanup()


debug exec (net:Page) =
InitContext2Petri() | [transform()] | ComposePage(net)
| composition_invariant_new()* | composition_invariant_new2() | Cleanup()
```

Normal Rules with Nested Alternatives

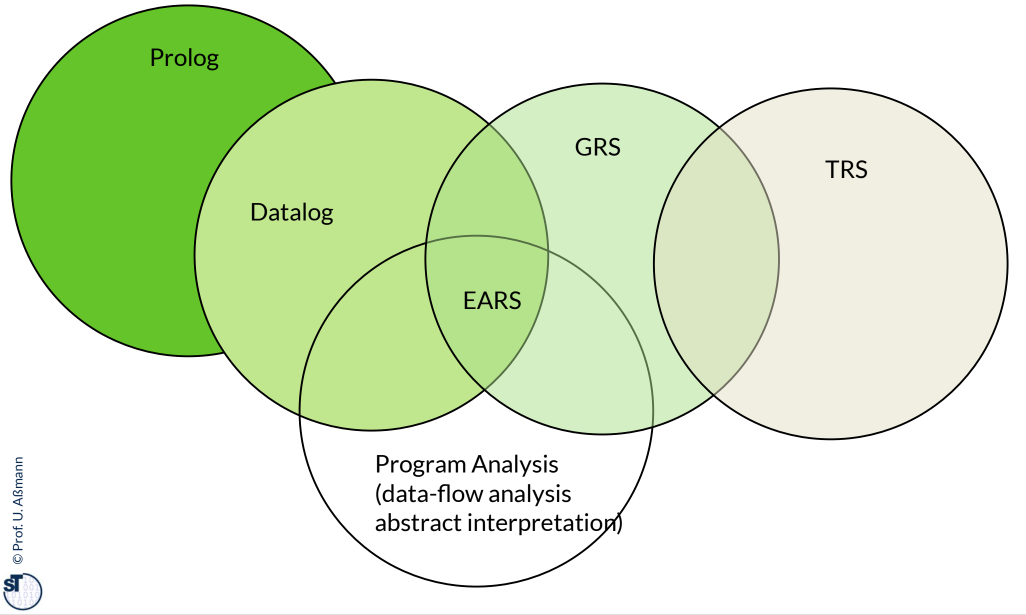

```
rule foldCond {
        cond:Cond -df0:Dataflow-> c0:Const;
        falseBlock:Block -falseEdge:False-> cond;
        trueBlock:Block -trueEdge:True-> cond;
  alternative {
     TrueCond {
        if { c0.value == 1; } modify {
                delete(falseEdge);
                -jmpEdge:Controlflow<trueEdge>->; }
      }
      FalseCond {
        if { c0.value == 0; } modify {
                delete(trueEdge);
                -jmpEdge:Controlflow<falseEdge>->; }
        }
      }
   }
   modify { delete(df0); jmp:Jmp<cond>; }
}
```

42.5 More on the Logic-Graph Isomorphism

Results

- Theory: Graph rewriting, DATALOG and data-flow analysis have a common core:
  - EARS build the bridge between graph analysis and transformations
- Exhaustive GRS: If a termination graph can be identified, a GRS terminates.
- Program optimization:
  - Spezification of program optimizations is possible with graph rewrite systems. Short specifications, fewer effort.
  - Practically usable optimizer components can be generated.
- Uniform Specification of Analysis and Transformation
  - If the program analysis (including abstract interpretation) is specified with GRS, it can be unified with program transformation
- Limitations
  - Several optimizations can be specified with GRS which are not exhaustive (peephole optimization, constant propagation with partial evaluation).
  - As general rule embedding is not allowed, a rule only matches a fixed number of nodes.
  - Thus those transformations, which refer to an arbitrary set of nodes, cannot be specified.

The Common Core of Logic, Rewriting and Program Analysis



An Old Citation

There clearly remains more work to be done in the following areas:

- discovery of other properties of transformations that appear to have relevance to code optimization,
- development of simple tests of these properties, and
- the use of these properties to construct efficient and effective optimization algorithms that apply the transformations involved.

Aho, Sethi, Ullmann in Code Optimization and Finite Church-Rosser Systems, 1972

The End

- Explain a connected graph pattern for local graph rewriting. What is a disconnected graph pattern for context-sensitive graph rewriting?
- What does it mean when GRS are exhaustive (XGRS)?

- Many GrGen examples are from Carl Mai
  - https://petrinets.pages.st.inf.tu-dresden.de/adaptive-petrinets/index.html
  - https://git-st.inf.tu-dresden.de/adaptive_petrinets/reconfnet

42.4. Model Transformations with ATL

ATLAS Transformation Language (ATL)
http://www.eclipse.org/atl/

Tools for Model-Driven Software Development

- In MDSD and MDA, horizontal and vertical model transformations should be specified with graph rewrite systems
- Example tools:
  - JastAdd RAGs (Java)
  - GrGen (C#)
  - **ATL** in Eclipse EMOF

ATL Integrates OCL as Query Language


```
// Transitive Closure  in ATL, with a recursive OCL query
rule computeTransitiveClosureBaseCase {
  from node: Node (
    // possible to call OCL expressions
    node->baserelation.collect( e | e.baserelation)->flatten() );
  )
  to newNode mapsTo node (
    // set new transitive relation
    newNode->transitiverelation <- node->baserelation
  )
}
rule computeTransitiveClosureRecursiveCase {
  from node: Node (
    node->transitiverelation.collect( e | e.baserelation)->flatten() );
  )
  to newNode mapsTo node (
    // set new transitive relation
    newNode->transitiverelation <- node->transitiverelation
  )
}
```

Terminology for Automated Graph Rewriting

- **Graph rewrite rule:** rule (left, right hand side) to match left-hand side in the graph and to transform it to the right-hand side
- **Graph rewrite system:** set of graph rewrite rules
- **Start graph (axiom):** input graph to rewriting
- **Graph rewrite problem:** a graph rewrite system applied to a start graph
- **Manipulated graph (host graph):** graph which is rewritten in graph rewrite problem
- **Redex**: (reducible expression) application place of a rule in the manipulated graph
- **Derivation:** a sequence of rewrite steps on the manipulated graph, starting from the start graph and ending in the normal form
- **Normal form:** result graph of rewriting; manipulated graphs without further redex
- **Unique normal form:** unique result of a rewrite system, applied to one start graph
- **Terminating GRS:** rewrite system that stops after finite number of rewrites
- **Confluent GRS:** two derivations always can be commuted, resp. joined together to one result
- **Convergent GRS:** rewrite system that always yields unique results (terminating and confluent)