25. Deep Model Analysis: Model and Program Analysis with Graph Reachability

Prof. Dr. Uwe Aßmann

Softwaretechnologie

Technische Universität Dresden

Version 15-0.9, 02.01.16

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

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Other Literature

- [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

25.1 Using EARS for Deep Analysis of Models and Mappings of Models and Code

- 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 relations for remotely reachable parts of a graph and a model
  - They **abbreviate long** paths in models
- EARS can be used for reachability and model mapping:
  - Transitive closure
  - Regular path reachability
  - Context-free path reachability

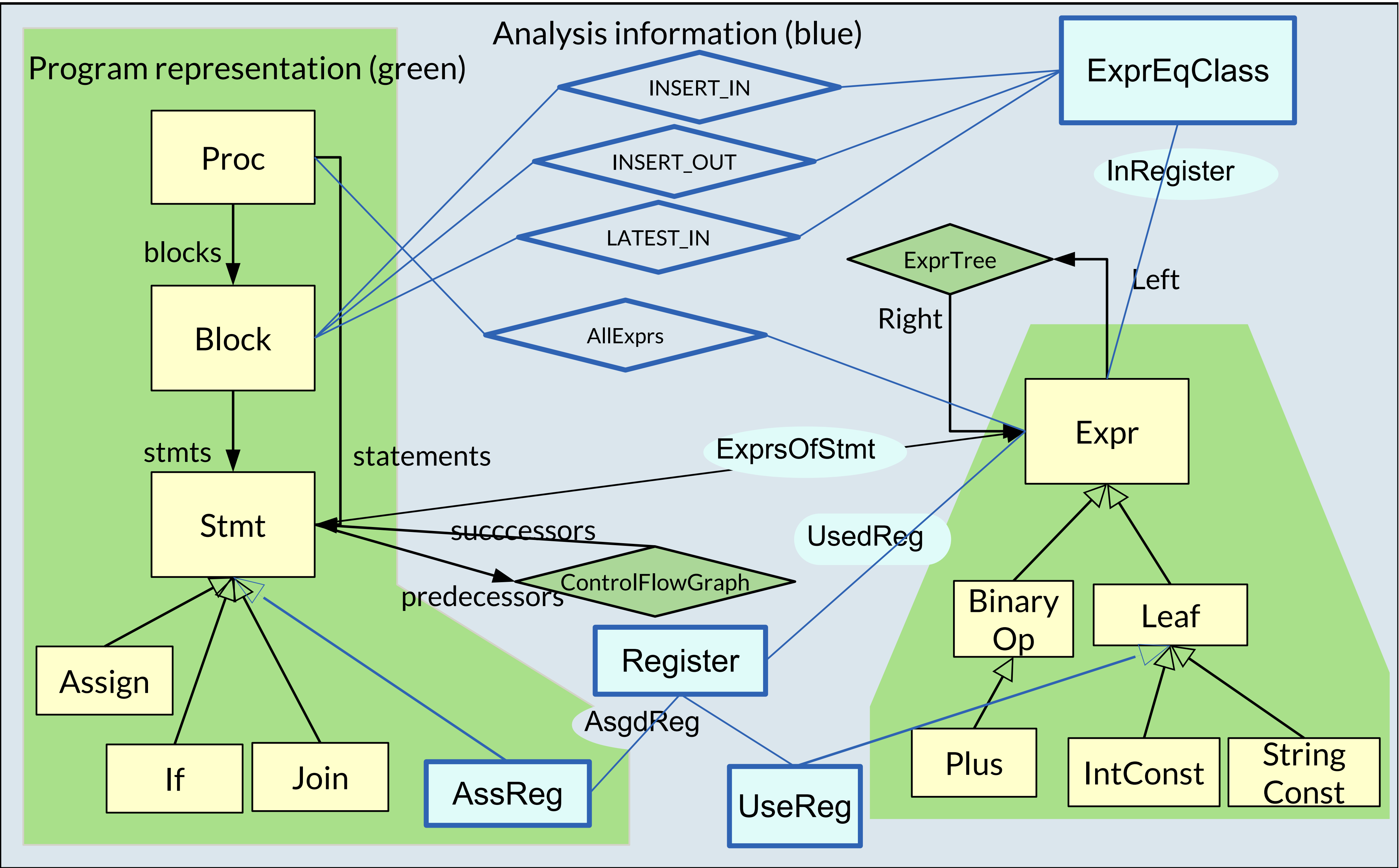
Model Analysis with Graph Reachability

- 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 addition rewrite systems (EARS) 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
- Use the graph-logic isomorphism to encode
  - Facts in graphs
  - Logic queries in graph rewrite systems

Specification Process

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 infered predicates over the program objects) as objects or relationships
2) **Flat model and program analysis** (preparing the abstract interpretation)
   - Querying graphs, enlarging graphs
   - Materializing implicit knowledge to explicit knowledge
3) **Deep model and program analysis**
   - Reachability
   - Inter-model reachability (traceability), materializing model mappings
4) **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
5) **Model and Program transformation** (optimization)
   - Transforming the program representation

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



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

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

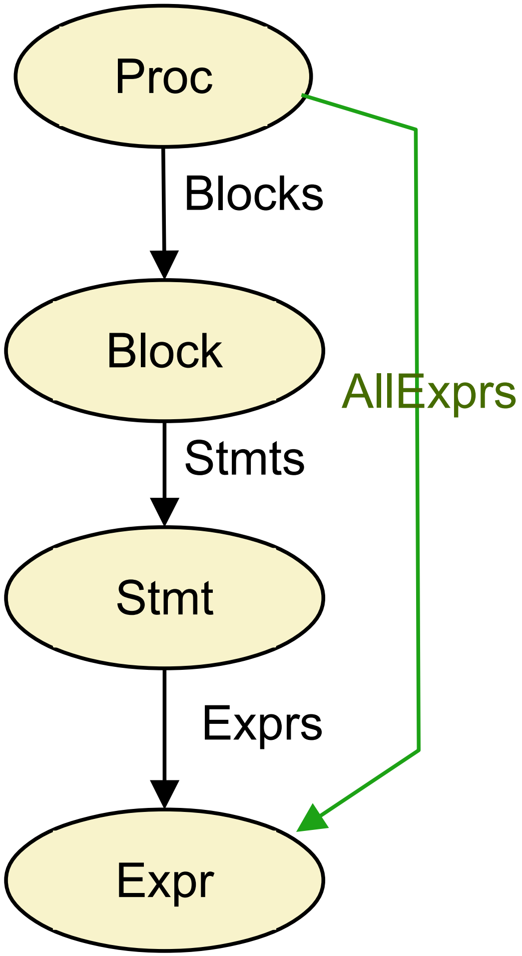
25.2.1. Simple Reachability of Model Elements and Models: Path Abbreviations in Graph Analysis

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

Path Abbreviations for Simple Reachability

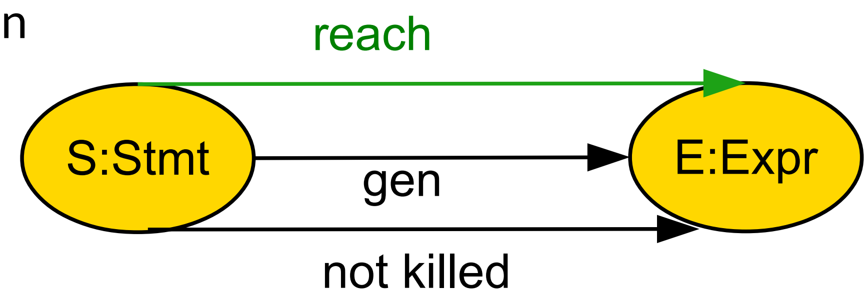
- Path abbreviations shorten paths in the manipulated graph.
- They may collect nodes into the neighborhood of other nodes.
- Ex.: Collection of Expressions for a procedure: edge addition

```
-- F-Datalog notation:
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.Exprs Expr
```

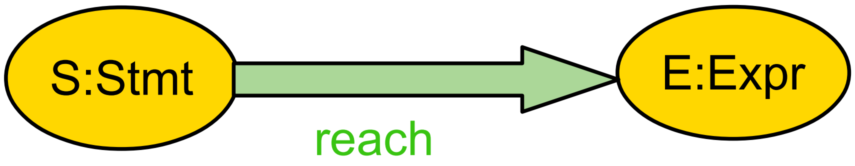


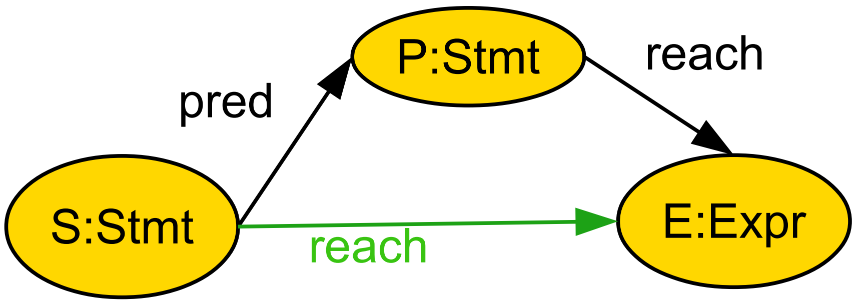
Transitive Closure (TC) for Remote Reachability

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




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



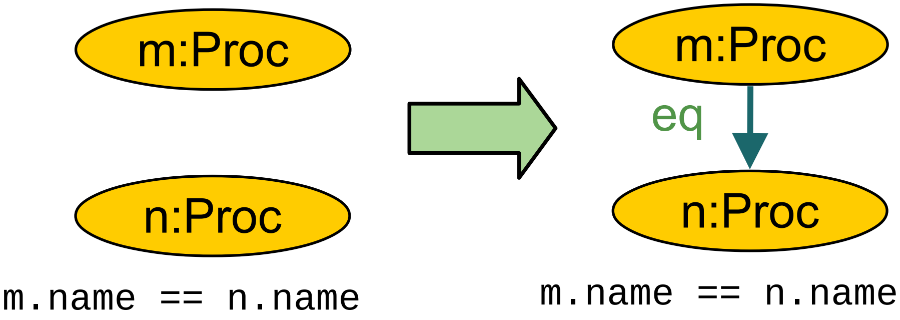


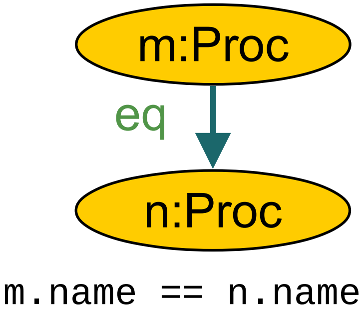

```
// F-Datalog
reach(S:Stmt,E:Expr) :- gen(S:Stmt,E:Expr), not killed(S:Stmt,E:Expr).
reach(S:Stmt,E:Expr) :- pred(S:Stmt,P), reach(P,E:Expr).
```

Ex.: Relating Nodes into Equivalence Classes

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

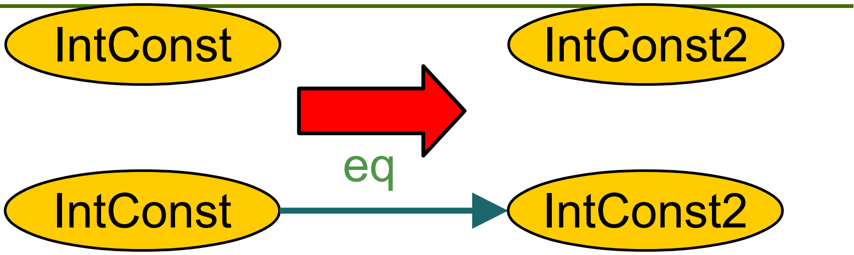
```
baserule:
eq(m:Proc,n:Proc) :-
   m.name == n.name.

–
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
```



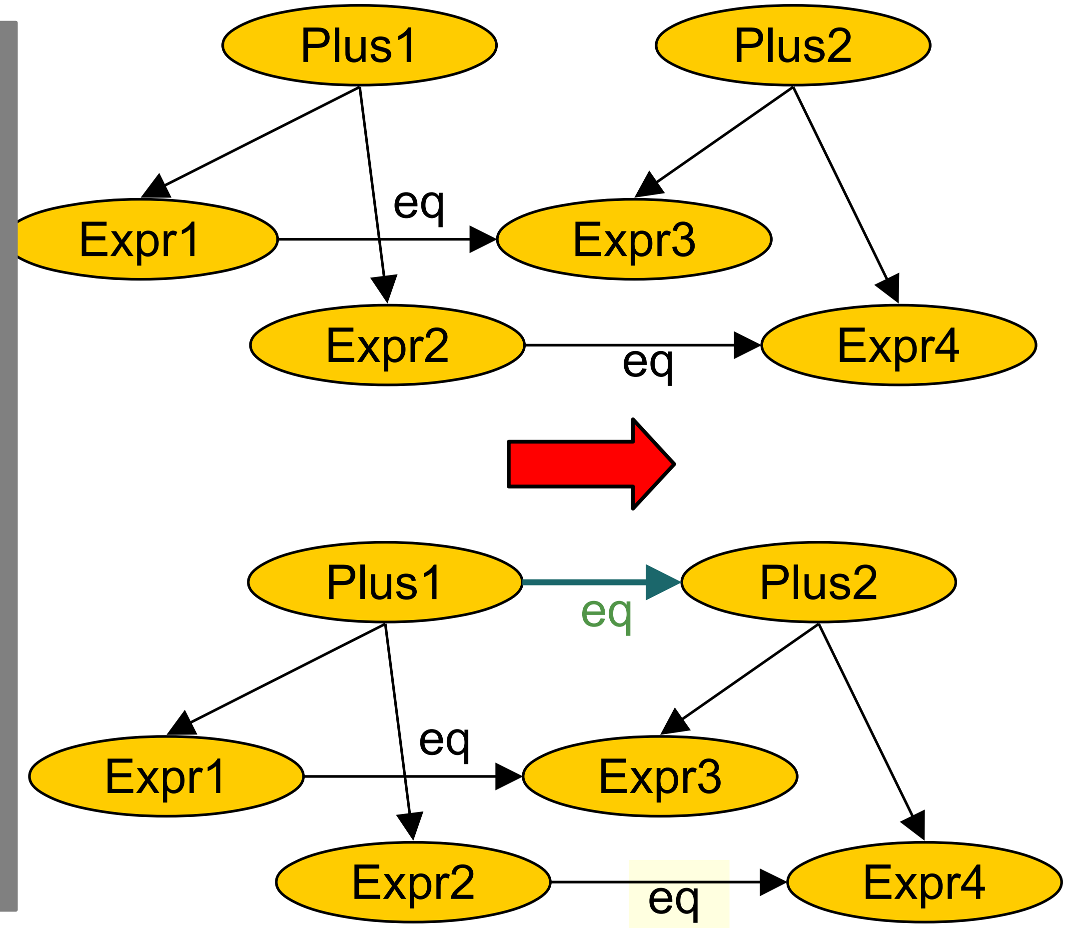


Ex. Relating Nodes into Equivalence Classes (Here: Value Numbering, Synt. Expression Equivalence)

- Ex.: Computing structurally equivalent expressions
- Question: "Which expression trees have the same structure?"




```
--- F-Datalog baserule:
eq(IntConst1,IntConst2) :-
   IntConst1 ~ IntConst(Value),
   IntConst2 ~ IntConst(Value).
--- recursive_rule:
eq(Plus1,Plus2) :-
   Plus1 ~ Plus(Type),
   Plus2 ~ Plus(Type),
   Left(Plus1,Expr1),
   Right(Plus1,Expr2),
   Left(Plus2,Expr3),
   Right(Plus2,Expr4).
   eq(Expr1,Expr3),
   eq(Expr2,Expr4).
```



25.3. Deep Model Analysis (Value-Flow Analysis, Data-Flow Analysis) as General Graph Reachability

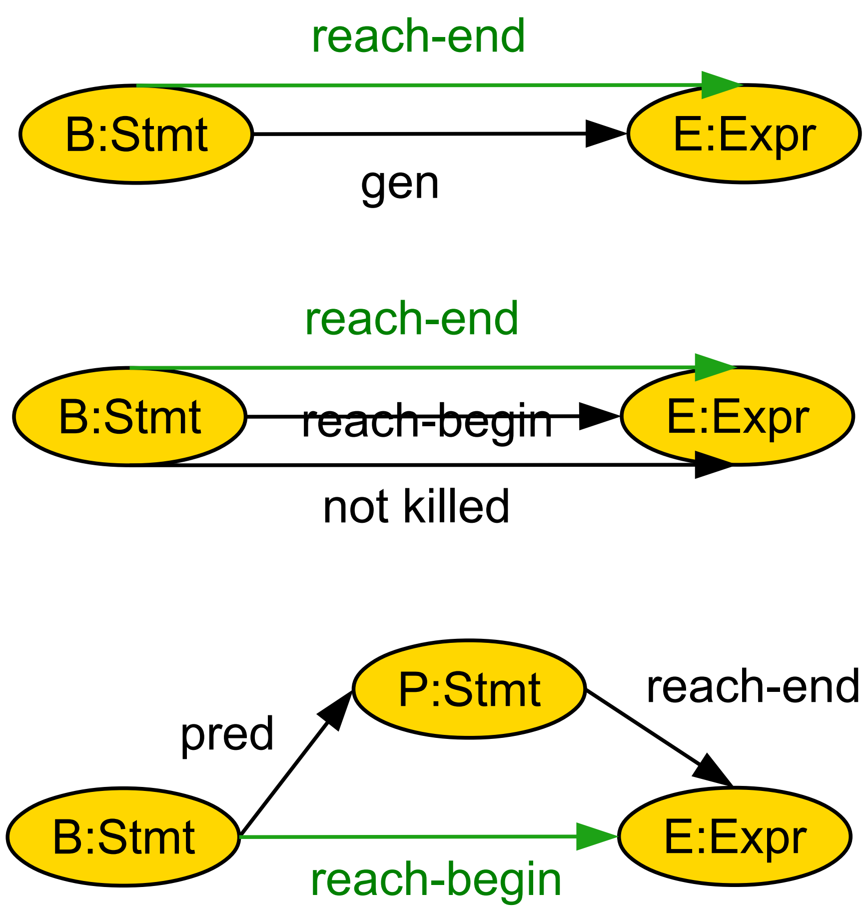
- with edge addition rewrite systems and F-Datalog

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
  - **Reaching Definitions Analysis:** Which Assignments (Definitions) of a variable can reach which statement?
  - **Live Variable Analysis:** At which statement is a variable live, i.e., will further be used?
  - **Busy Expression Analysis:** Which expression will be used on all outgoing paths?
  - Central part: 1 recursive system

Reaching Definition Analysis By Abstract Interpretation with EARS

- **Problem:** "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-begin)
  - (B reach-end E) := (E reaches end of block B)
-



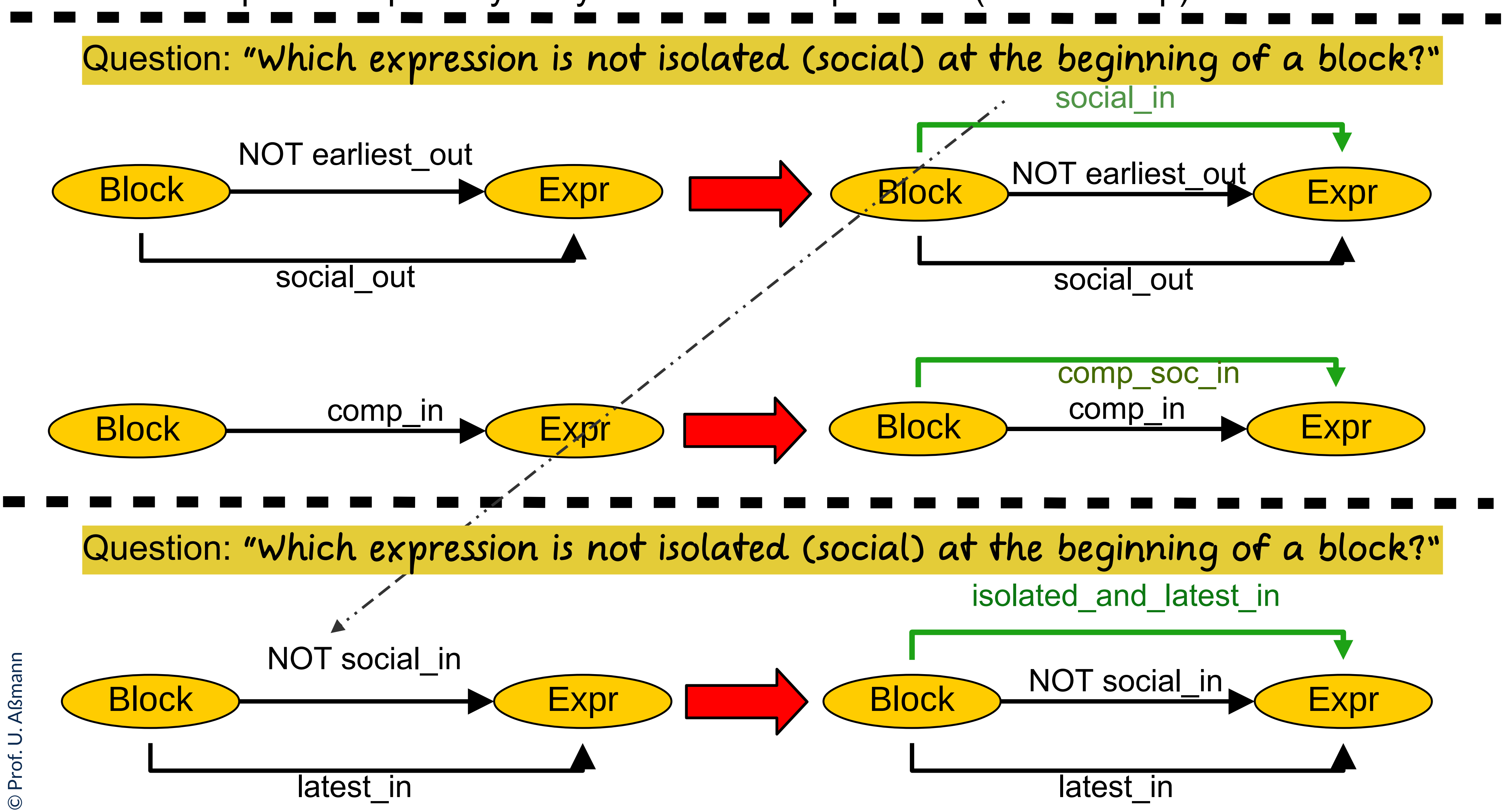

```
reach-end(B,E) :- gen(B,E).
reach-end(B,E) :- reach-begin(B,E), not killed(B,E).
reach-begin(B,E) :-pred(B,P), reach-end(P,E).
```

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)

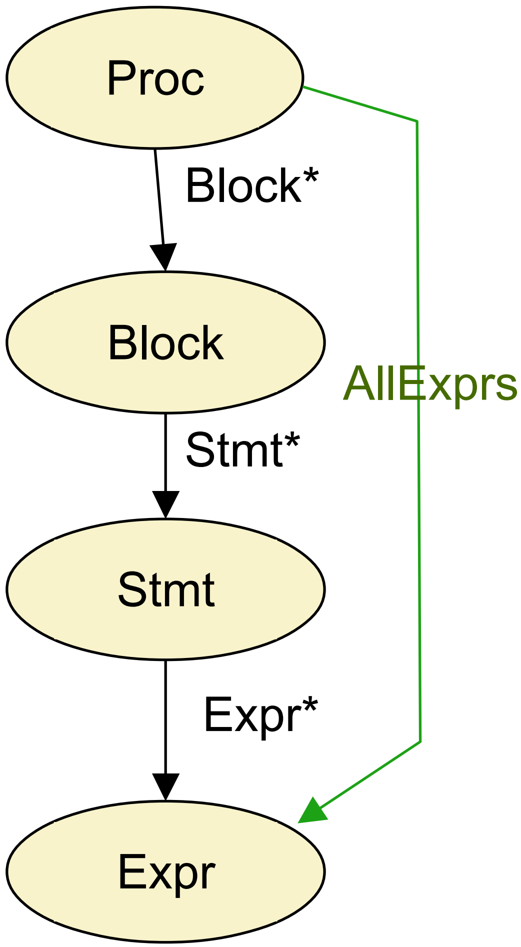


25.3.2 Regular Graph Reachability and Slicing

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).
-- if-then rules:
if  Block*(Proc,Block),
    Stmt*(Block,Stmt),
    Expr*(Stmt,Expr)
then
   AllExprs(Proc,Expr);
– regular expression notation (TGreQL):
AllExprs := Proc Block*.Stmt*.Expr* Expr
```



Static Slicing: Single-Source-Multiple-Target Regular Reachability

- [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 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

25.3.3 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

25.3.4 More on the Logic-Graph Isomorphism

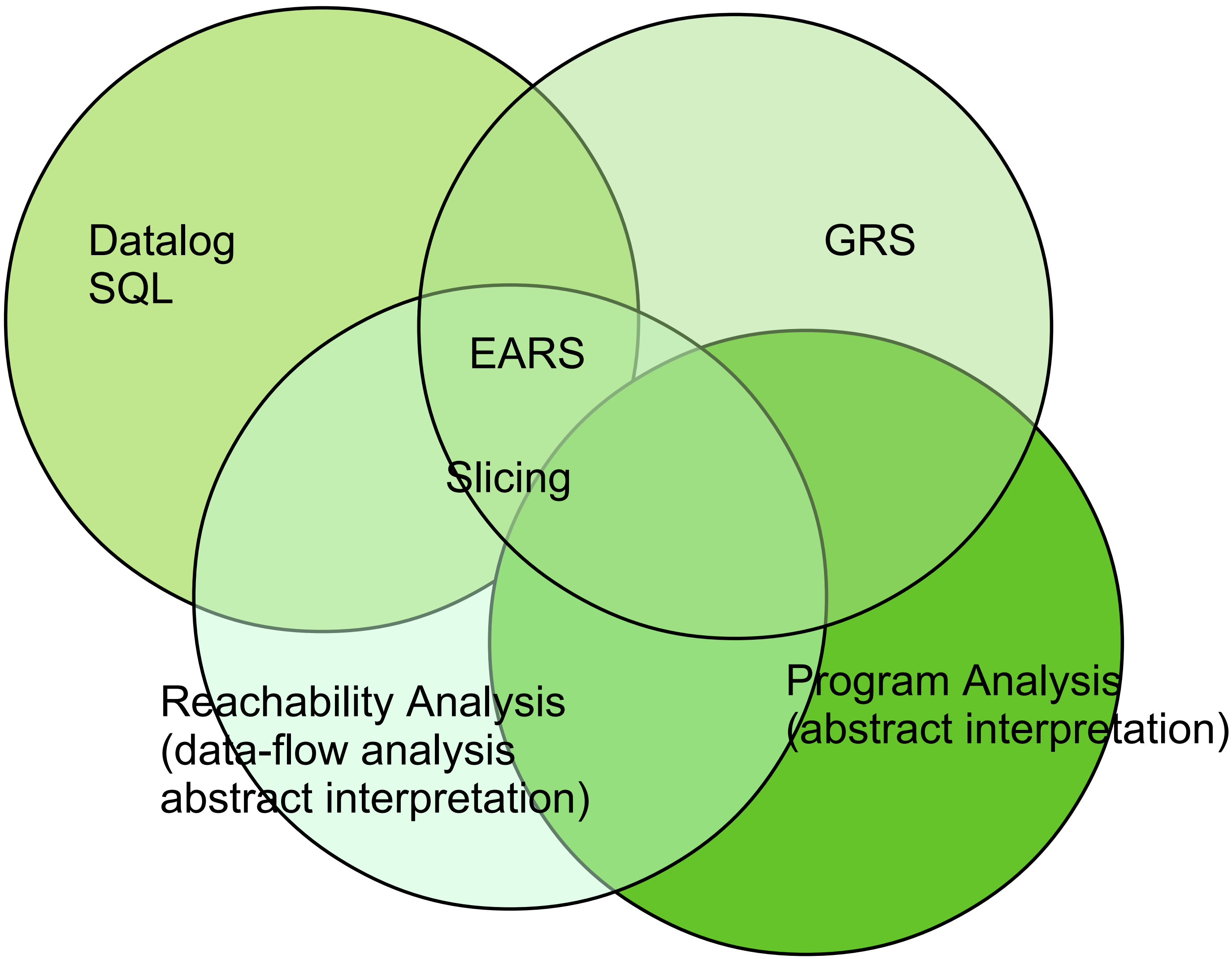
- [Courcelle] discovered that many problems can be expressed in logic (on facts) and in graph rewriting (on graphs)

Program and Model Analyses Covered by Graph Reachability

- Graph Reachability Analysis can do abstract interpretation
  - If it adds analysis information to the 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
  - 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



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

25.3.5 Implementation of Data-Flow Analysis in Tools

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

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

Graph Rewrite Tools for Graph Reachability

- 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 iInterpreters
  - http://groove.cs.utwente.nl/

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

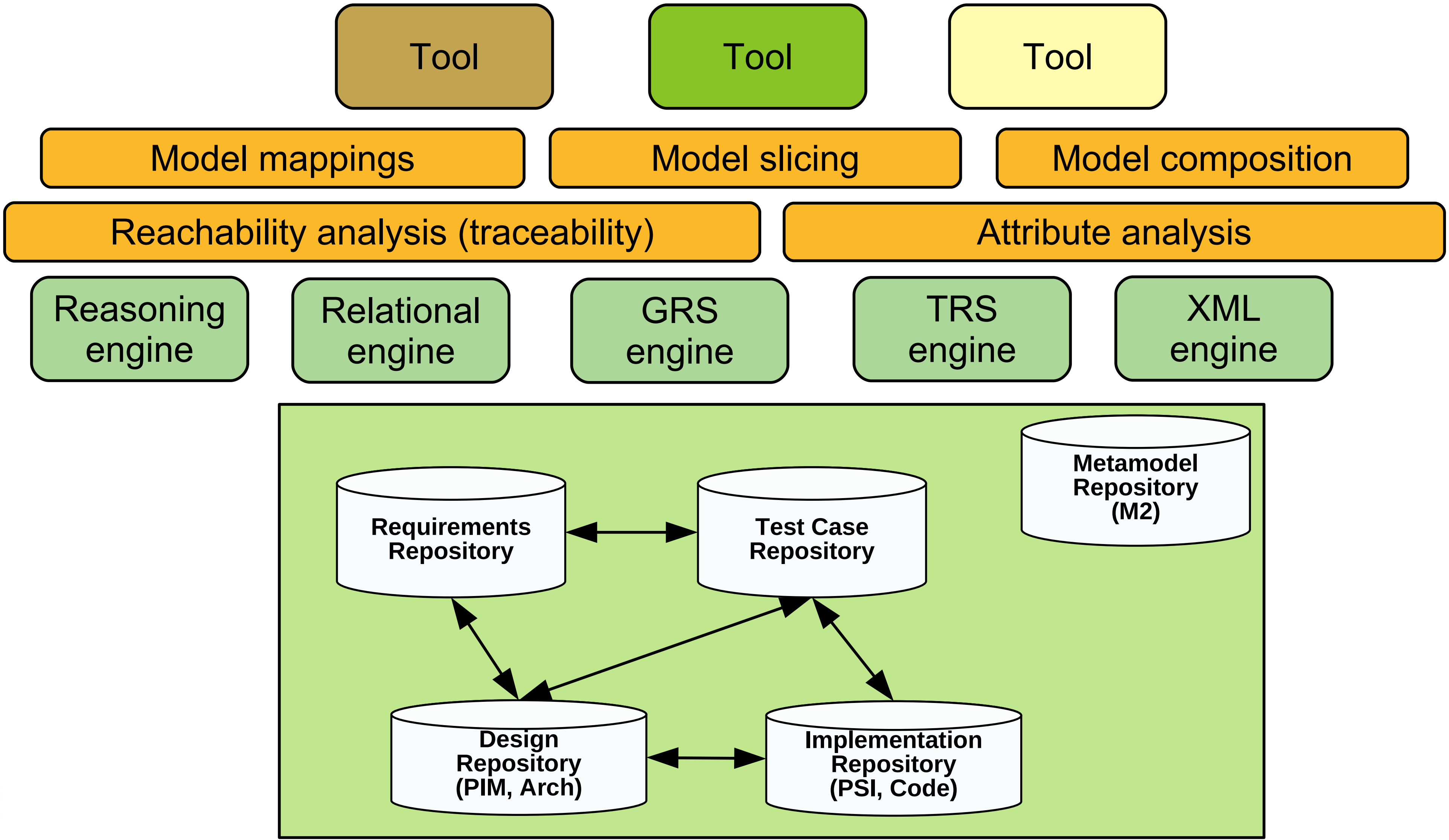
- Model mapping languages are model query languages who enter their results again into the models as analysis information.
- They create *model mappings* which are important for macromodels.

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.

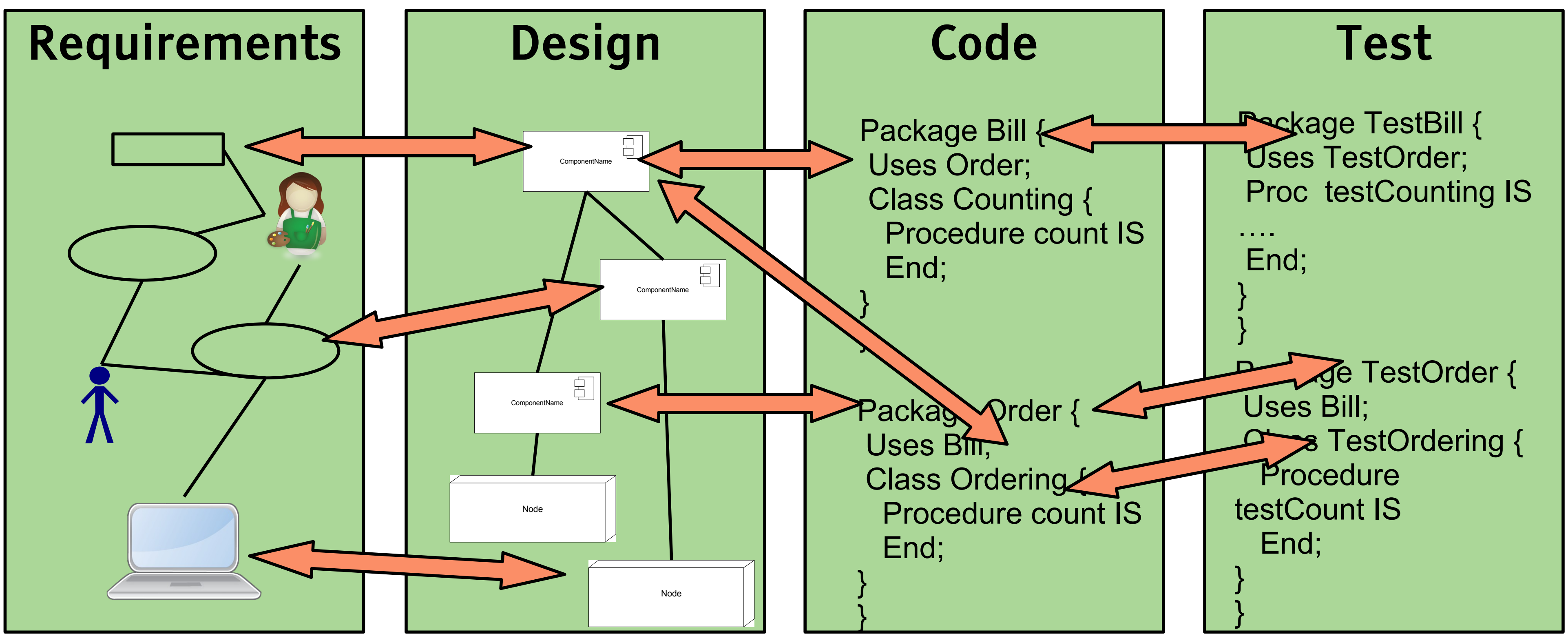
Q2: Tools in an Integrated Development Environment (IDE) for MDSD

- **Model mappings** relate different models to enable **reachability analysis, trace analysis** (if models are in different repositories) and **impact analysis**
- An **in-memory macromodel** is a megamodel where all models are loaded in memory



Q12: The ReDeCT Problem and its Macromodel

- The **ReDeCT problem** is the problem how requirements, design, code and tests are related (? V model)
- Mappings between the Requirements model, Design model, Code, Test cases
- A **ReDeCT macromodel** has maintained mappings between all 4 models
- If all models belong to one repository, we call it a **mono-repository macromodel**
- If the models belong to multiple repositories, we call it a **multi-repository macromodel**
  - **Then, Reachability means Traceability**



Advantages of Model Mappings

- **Error tracing**
  - When an error occurs during testing or runtime, we want to trace back the error to a design element or requirements element
- **Traceability**
  - We want to know which requirement (feature) influences which design, code, and test elements, so that we can demarcate modules in the solution space (product line development)
- **Synchronization in Development:**
  - Two models are called **synchronized**, if the change of one of them leads automatically to a hot-update of the other
- **Cohesion of Distributed Information:**
  - Two related model elements may contain distributed information about a thing. The relation allows for reconstructing the full information
  - Example:
    - Storing two roles of an object in two different models (See "Amoeba Object Pattern")
    - Splitting the representation of the requirements on an object and its design in requirements vs design model

Different Forms of Model Mappings

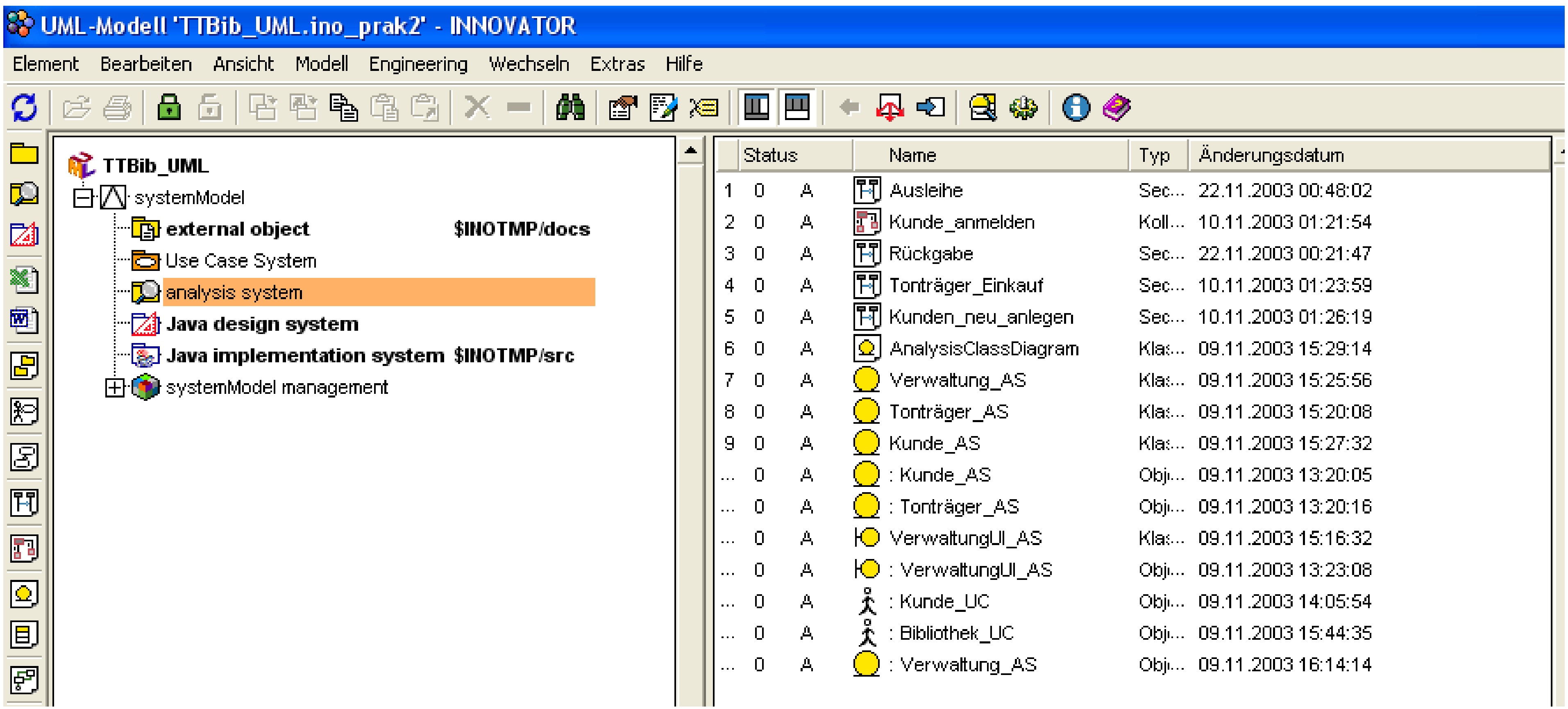
- **Directly specified mappings** specify a deterministic mapping function between a source and target model.
  - Direct mappings are specified in GUI or text files
  - Direct mappings may be ***complete*** or ***incomplete***
- **Recursive mappings** are defined in a functional language
  - **Denotational semantics** is a complete direct mapping of two languages
  - The **coverage** of the source model must be ensured (completeness of specification)
- **General mappings** may be intensionally specified. Source and target models are mapped
  - With graph reachability expressions (QVT-R, TgreQL, EARS)
  - With query expressions (Semmle.QL)
  - With expressions in a logic (F-Datalog)
- **Inter-model mappings** are defined between model elements of different models
- **Lifted inter-model mappings** are lifted from intra-model element mappings

25.4.1 Direct Mappings for Simple Traceability

- With a **direct model mapping**, a requirements model can be linked
  - to a test case specification
  - to a documentation
  - to an architectural specification
  - via the architectural specification, to the classes and procedures in the code

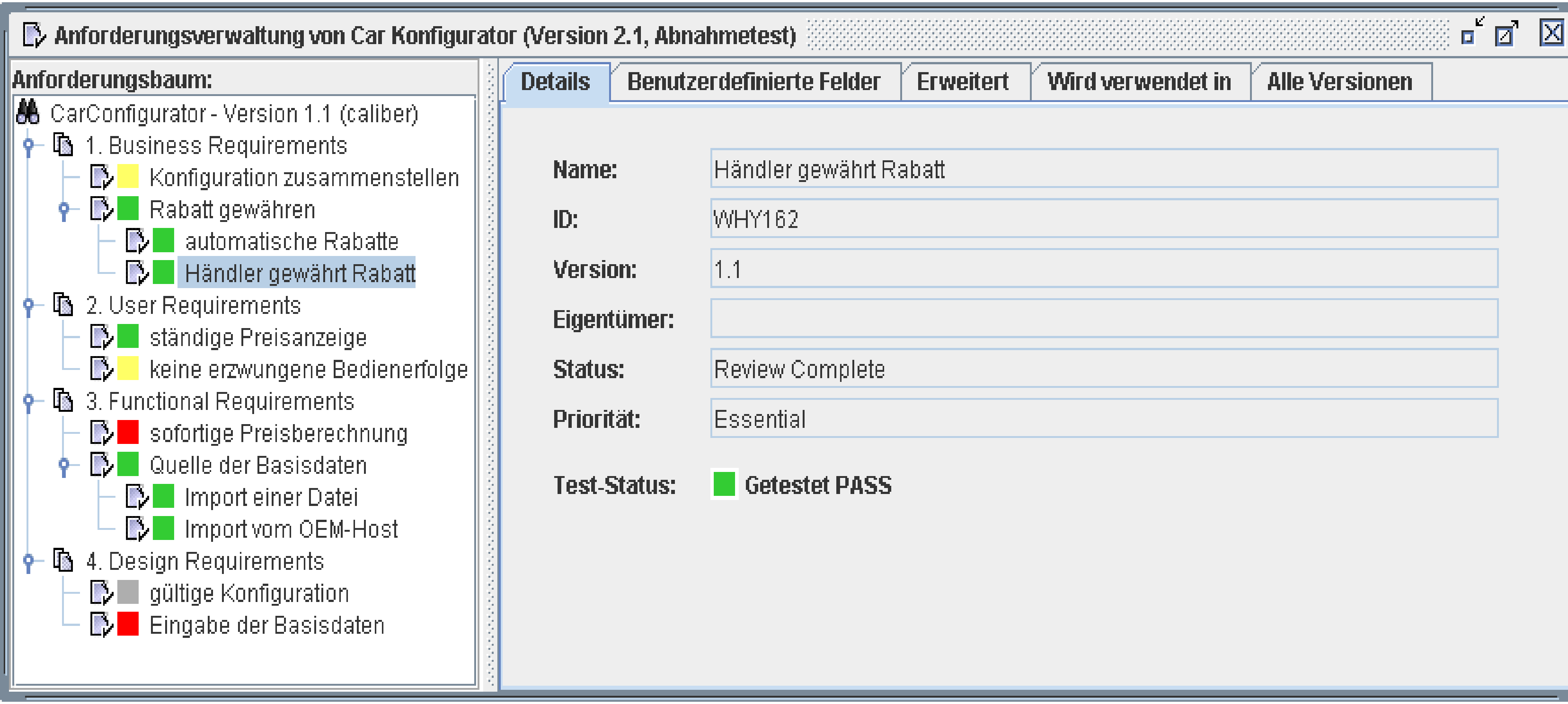
Ex.: Explicit Model Mapping (Modell-Verknüpfung) with MID INNOVATOR

- MID Innovator can be used for requirements models (use cases), design models, implementation models, as well as for transformations in between
- How to relate these models systematically?

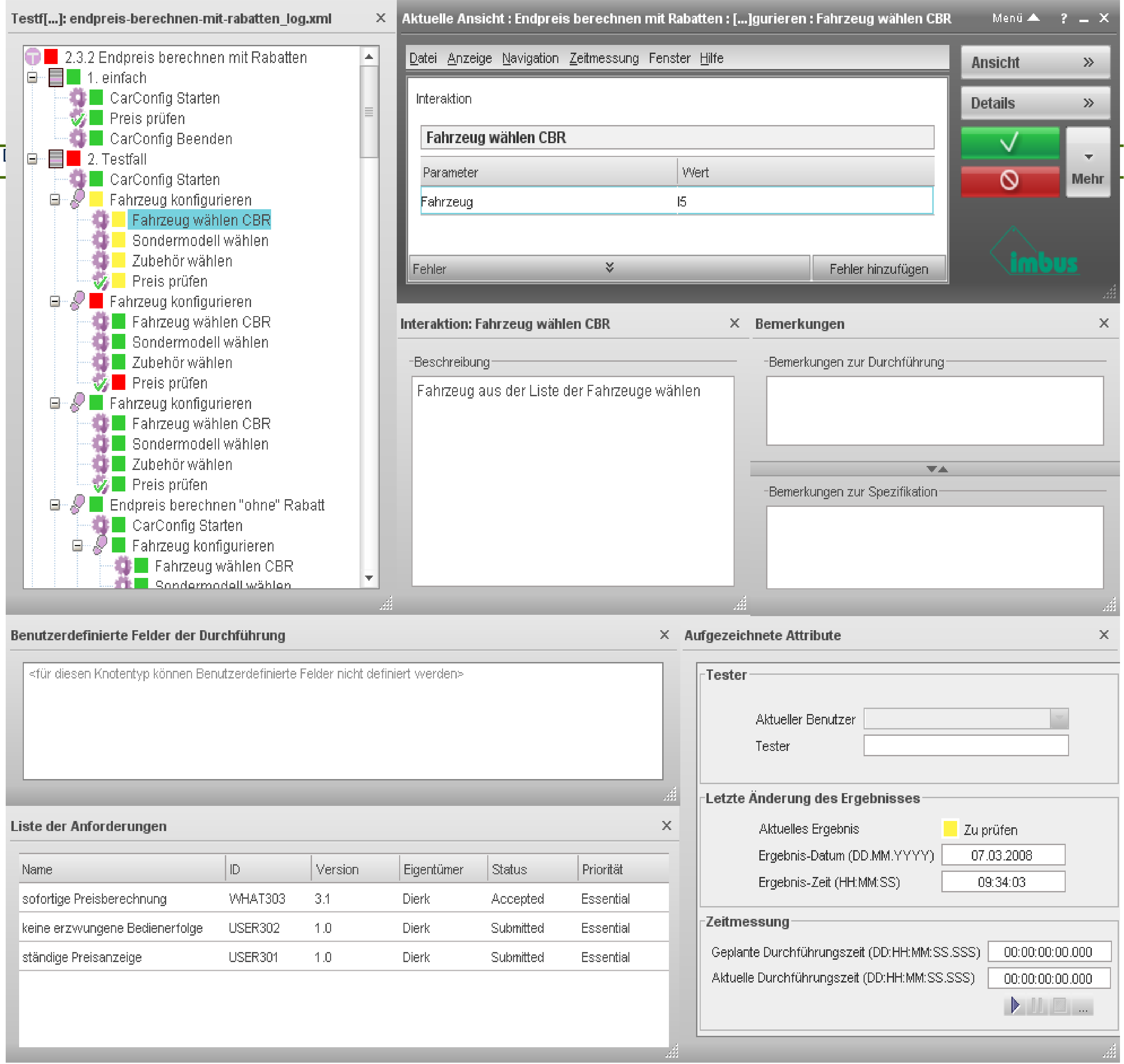


Example: imbus TestBench

Requirements get “red-yellow-green” Test Status Attribute



http://www.imbus.de/produkte/imbus-testbench/hauptfunktionen/

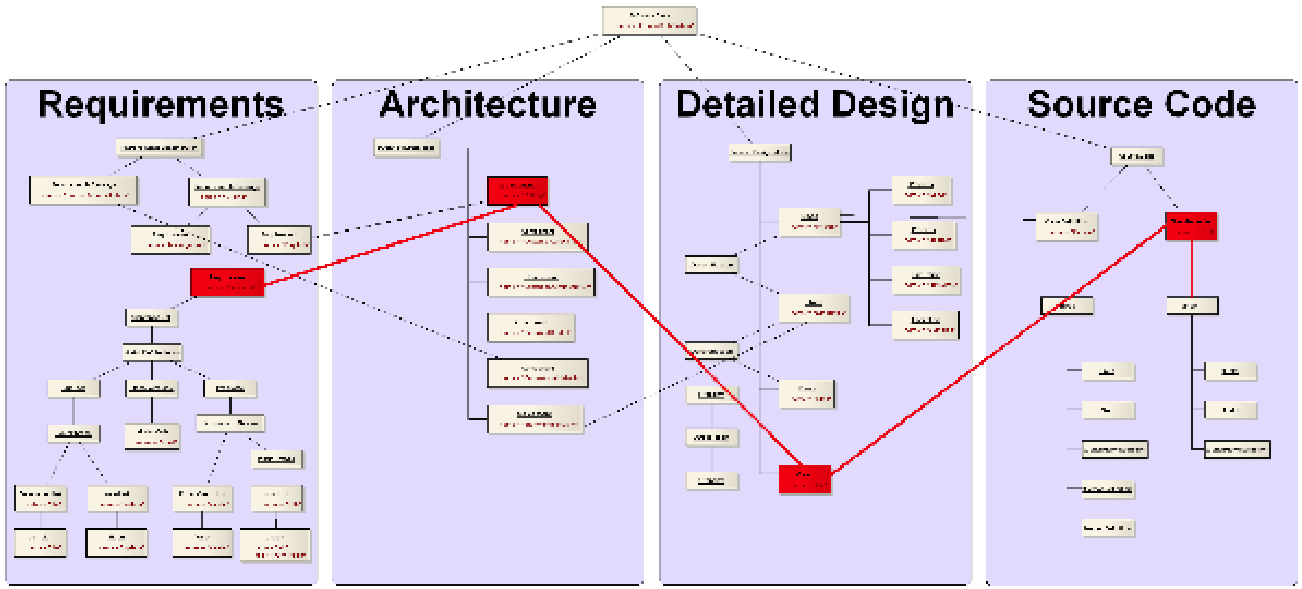

Testf[...]: endpreis-berechnen-mit-rabatten_log.xml
2.3.2 Endpreis berechnen mit Rabatten
1. einfach
CarConfig Starten
Preis prüfen
CarConfig Beenden
2. Testfall
CarConfig Starten
Fahrzeug konfigurieren
Fahrzeug wählen CBR
Sondermodell wählen
Zubehör wählen
Preis prüfen
Fahrzeug konfigurieren
Fahrzeug wählen CBR
Sondermodell wählen
Zubehör wählen
Preis prüfen
Fahrzeug konfigurieren
Fahrzeug wählen CBR
Sondermodell wählen
Zubehör wählen
Preis prüfen
Endpreis berechnen "ohne" Rabatt
CarConfig Starten
Fahrzeug konfigurieren
Fahrzeug wählen CBR
Aktuelle Ansicht : Endpreis berechnen mit Rabatten : [...]gurieren : Fahrzeug wählen CBR
Menü
Datei Anzeige Navigation Zeitmessung Fenster Hilfe
Ansicht
Details
Mehr
Interaktion
Fahrzeug wählen CBR
Parameter
Wert
Fahrzeug
I5
Fehler
Fehler hinzufügen
imbus
Interaktion: Fahrzeug wählen CBR
Beschreibung
Fahrzeug aus der Liste der Fahrzeuge wählen
Bemerkungen
Bemerkungen zur Durchführung
Bemerkungen zur Spezifikation
Benutzerdefinierte Felder der Durchführung
<für diesen Knotentyp können Benutzerdefinierte Felder nicht definiert werden>
Aufgezeichnete Attribute
Tester
Aktueller Benutzer
Tester
Letzte Änderung des Ergebnisses
Aktuelles Ergebnis
Zu prüfen
Ergebnis-Datum (DD.MM.YYYY)
07.03.2008
Ergebnis-Zeit (HH:MM:SS)
09:34:03
Zeitmessung
Geplante Durchführungszeit (DD:HH:MM:SS.SSS)
00:00:00:00.000
Aktuelle Durchführungszeit (DD:HH:MM:SS.SSS)
00:00:00:00.000
Liste der Anforderungen
Name
ID
Version
Eigentümer
Status
Priorität
sofortige Preisberechnung
WHAT303
3.1
Dierk
Accepted
Essential
keine erzwungene Bedienerfolge
USER302
1.0
Dierk
Submitted
Essential
ständige Preisanzeige
USER301
1.0
Dierk
Submitted
Essential

25.4.2. 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

25.4.2 Specifying Inter-Model Mappings with Model Mapping Languages

Ex.: Querying in ReDeCT

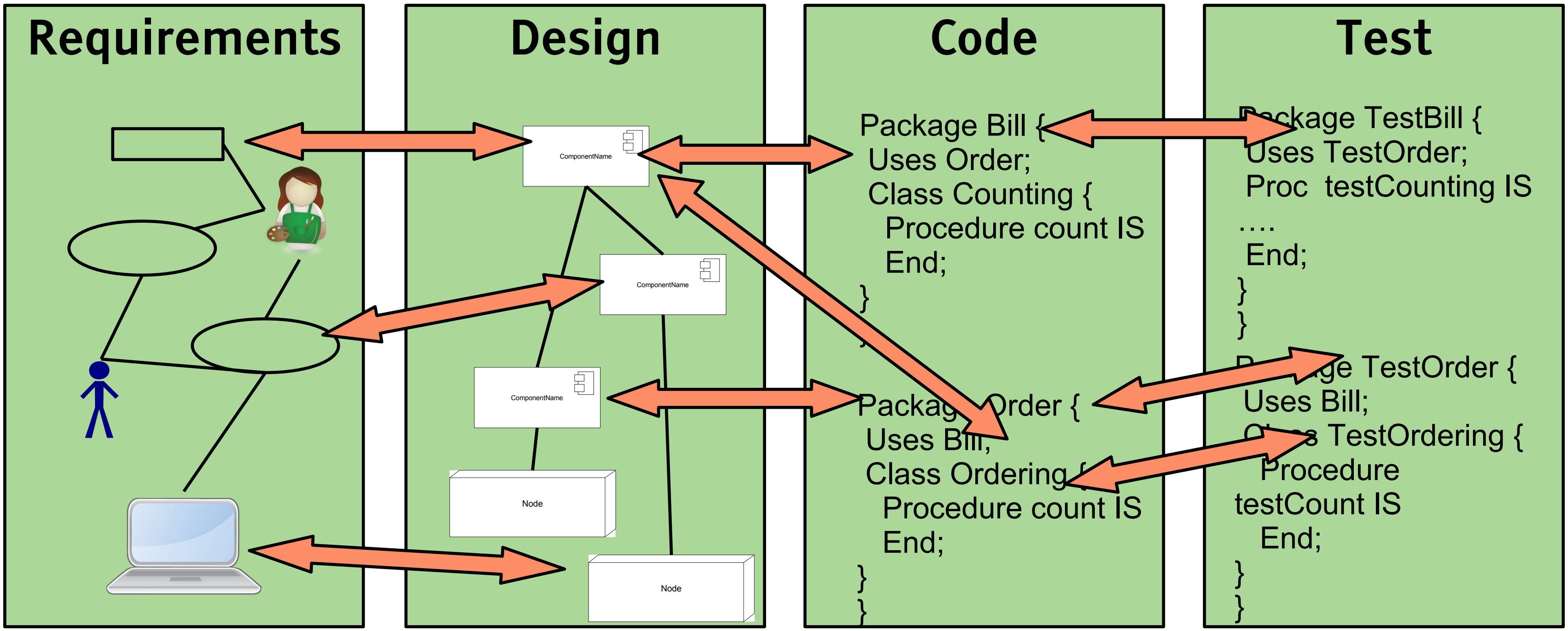



```
elementsIn(
  from req:V{Requirement}, archElem:V{UMLElement},
      desElem:V{UMLElement}, class:V{ClassDefinition}
  with req.name="Create bills" and
      req <--{Satisfies} archElem and
      archElem <--{Realize} desElem
      desElem <--{Implements} class
  report req, archElem, desElem, class
  end
)
```

Fig. 4. Sample GReQL query with associated slice of a software case

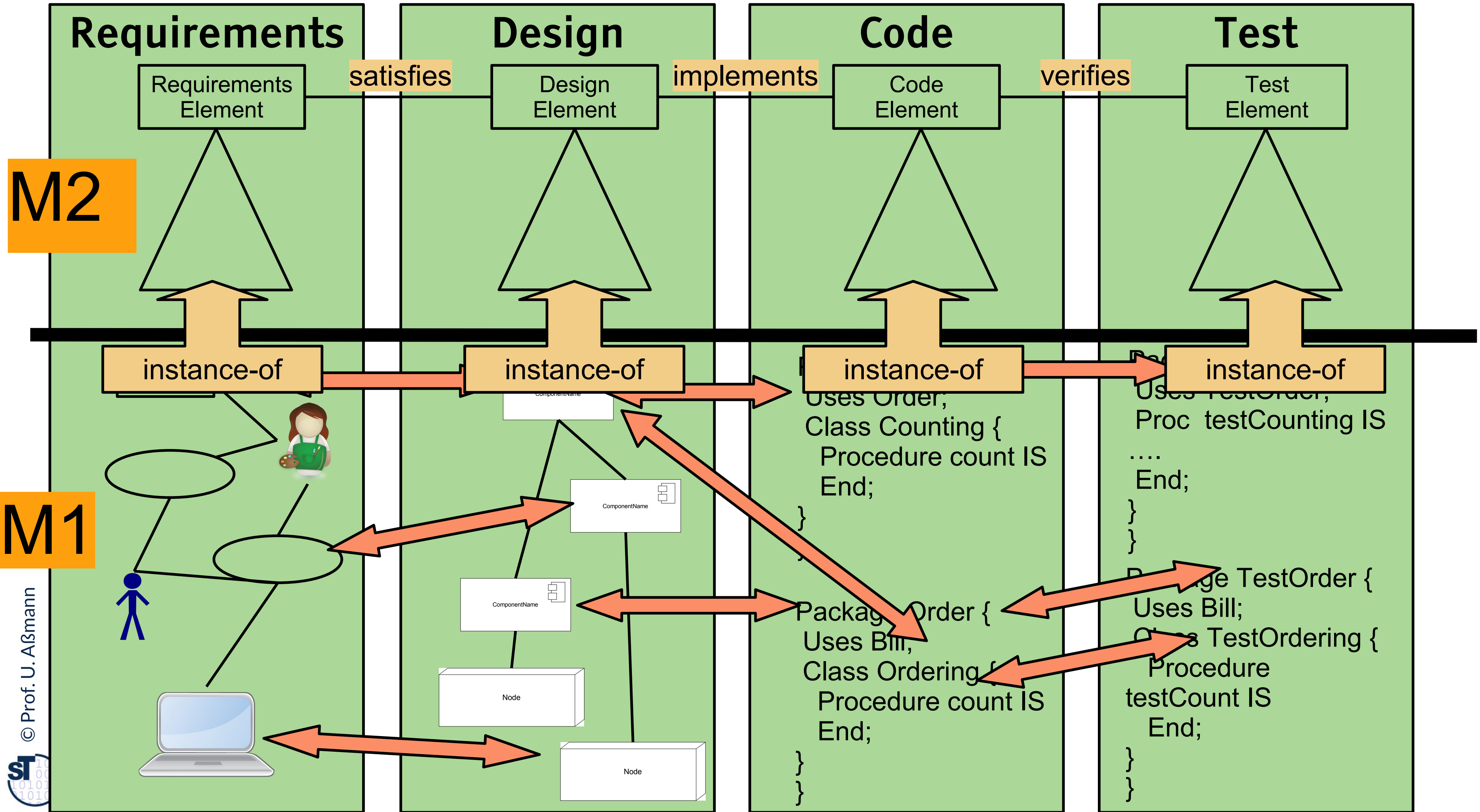
Inter-Model Relationships in The ReDeCT Macromodel

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



Inter-Model Relationships in The ReDeCT Macromodel

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



Specification of Traceability in ReDeCT with TGreQL

[BERS08]

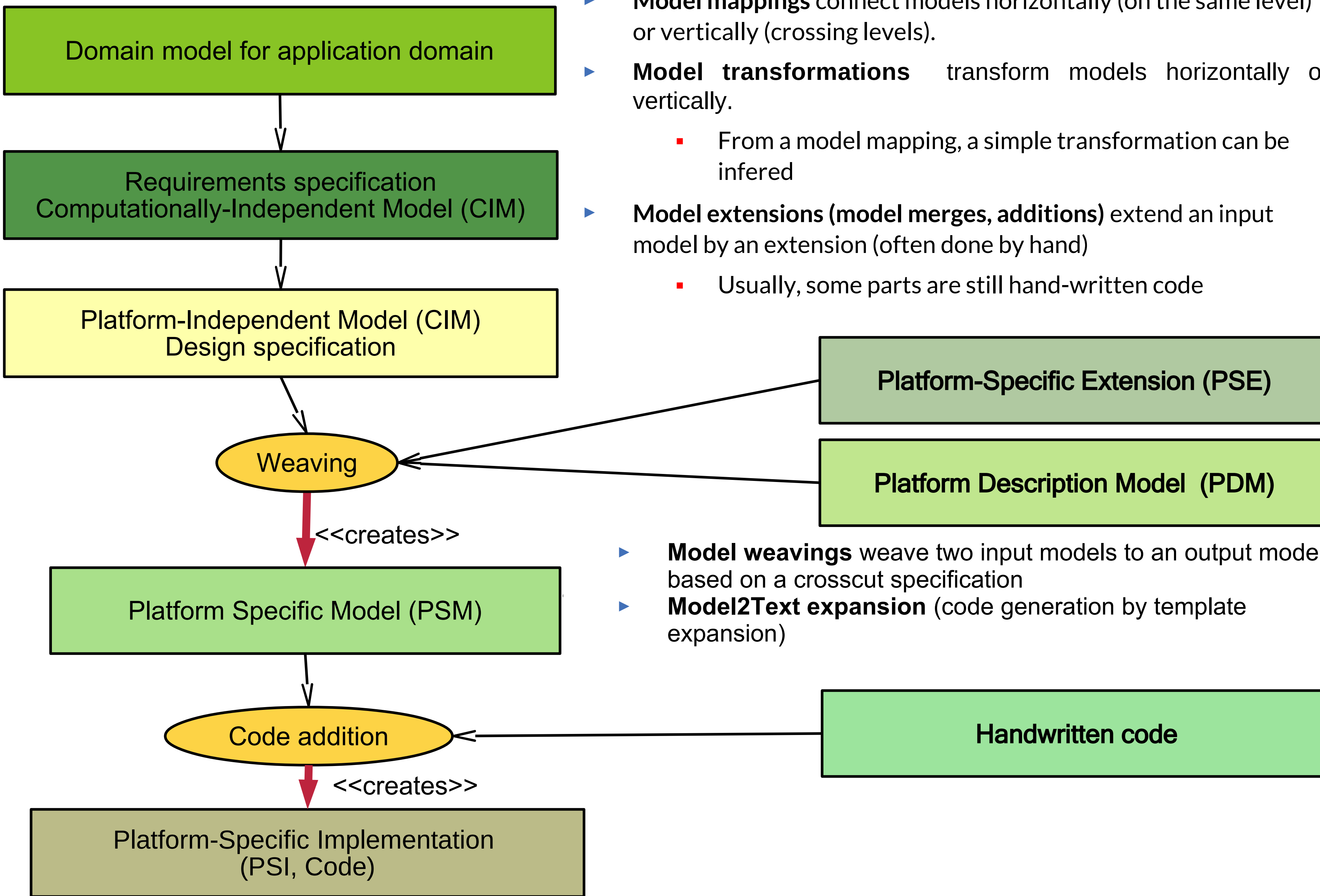
- **Direct inter-model relationships** form the basis of queries in the macromodel. Allow for the definition of
  - **Traceability relations** between model elements of different models
  - Hyperedges (tuples) between several model elements of different models
- **Any query language can be used for model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

```
// Defining a inter-model hyperedge (tuple) in TGreQL [BERS08]
elementsIn(
  from req:V{Requirement}, archElem:V{UMLElement},
   desElem:V{UMLElement}, 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
)
```

25.4.3 Inter-Model Reachability (Traceability)

- When models are kept in different repositories, inter-model reachability becomes *traceability*

Q9: Model Mappings and Model Weavings in the MDA Megamodel



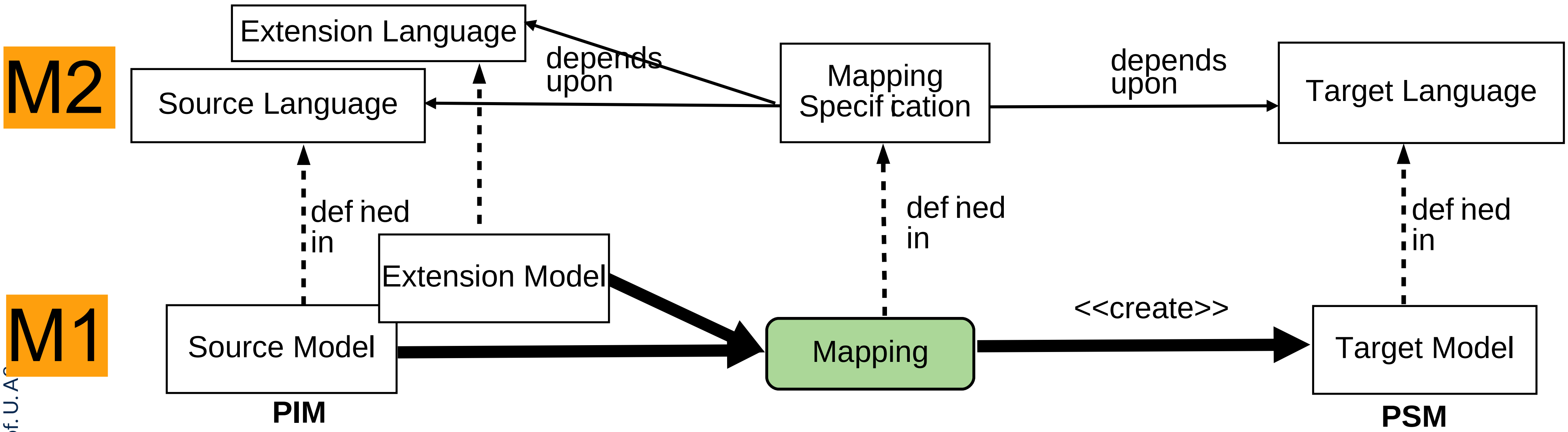
- **Model mappings** connect models horizontally (on the same level) or vertically (crossing levels).
- **Model transformations** transform models horizontally or vertically.
  - From a model mapping, a simple transformation can be infered
- **Model extensions (model merges, additions)** extend an input model by an extension (often done by hand)
  - Usually, some parts are still hand-written code

- **Model weavings** weave two input models to an output model, based on a crosscut specification
- **Model2Text expansion** (code generation by template expansion)

Rpt. from ST-II: Model Mapping, Transformation and Synchronization in the MDA

The MDA macromodel derives from a *platform-independent model* (**PIM) by hand, by rules, by transformations, by metaprograms** *platform-specific* models (**PSM)**

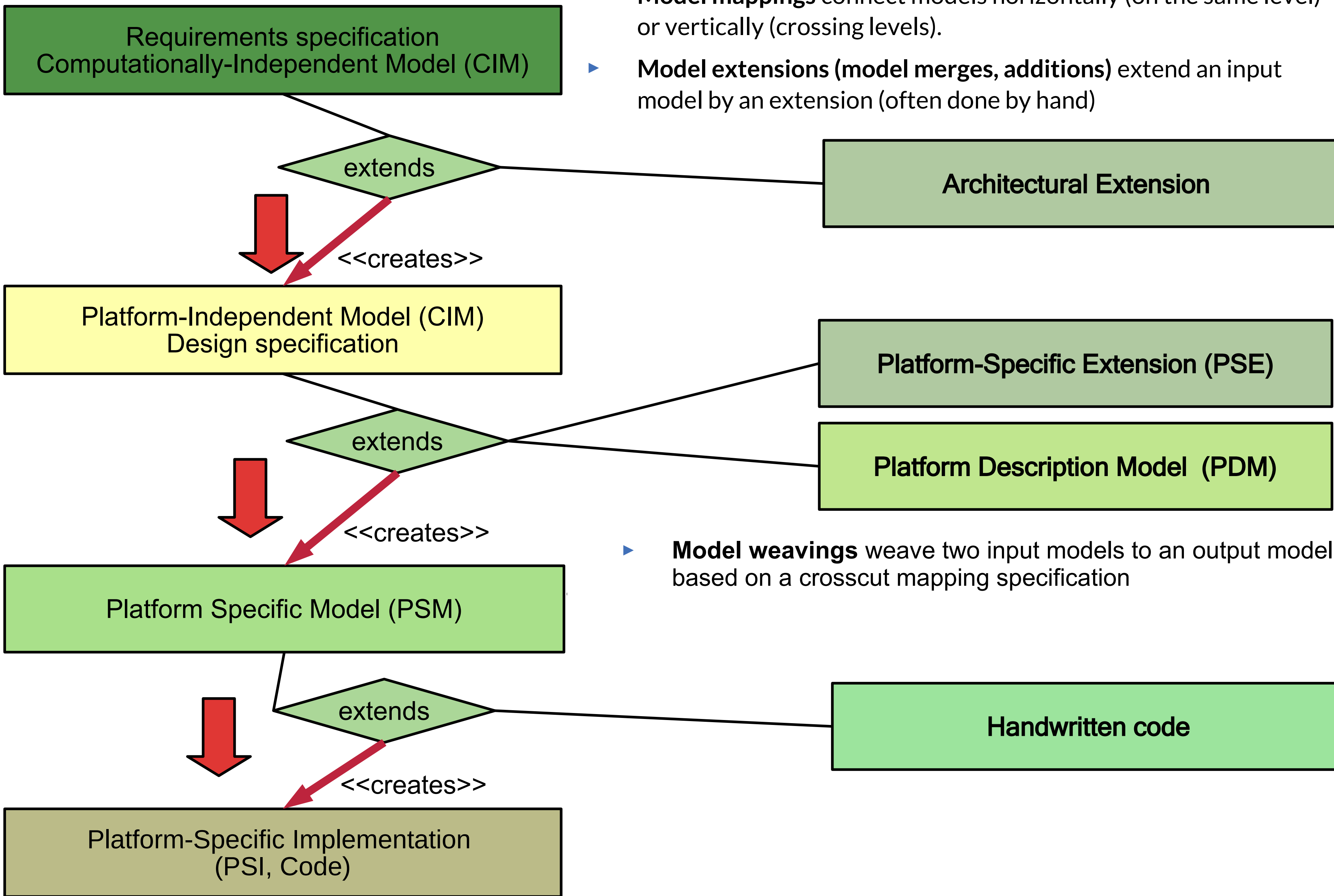
- Model mapping connects systematically all elements of a **source model** (in a **source language)** to the elements of a **target model** in a **target language.**
- From the mappings, a translation, transformation, or synchronization can be automatically infered.



Adapted from: Kleppe, A., Warmer, J., Bast, W.: MDA Explained - Practice and Promise of the Model Driven Architecture; Addison Wesley 2003 (Draft 25.10.02)

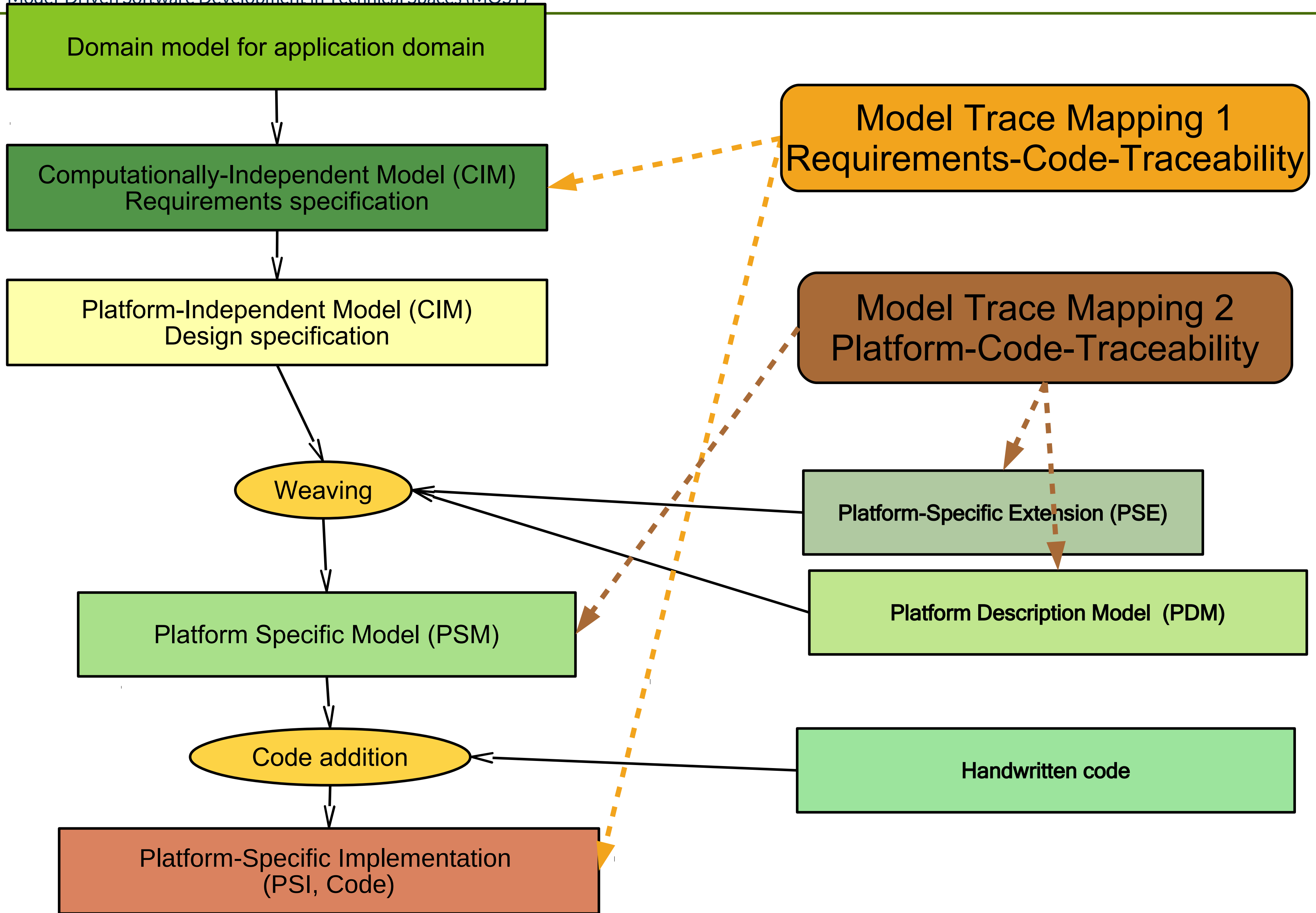
Q9b: Inter-Model Mappings in the MDA Megamodel

- **Model mappings** connect models horizontally (on the same level) or vertically (crossing levels).
- **Model extensions (model merges, additions)** extend an input model by an extension (often done by hand)



- **Model weavings** weave two input models to an output model, based on a crosscut mapping specification

Application of Traceability: Inter-Model Trace Mappings in the Macromodel MDA



The End - Appendix
Comprehension Questions

- 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?