TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

12) Validation of Graph-Based Models (Analysis and Consistency of Models)

- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediatechnik
- Gruppe Softwaretechnologie
- http://st.inf.tu-dresden.de/teaching/swt2
- Version 11-0.2, 21.11.12

Contents

- Different kinds of relations: Lists, Trees, Dags, Graphs
- Treating graph-based models – The graph-logic isomorphism
- Analysis, querying, searching graph-based models
    - The Same Generation Problem
    - Datalog and EARS
    - Transitive Closure
- Consistency checking of graph-based specifications (aka model validation)
    - Projections of graphs
    - Transformation of graphs

Obligatory Reading

- Jazayeri Chap 3
- If you have Balzert, Macasziek or Pfleeger, read the lecture slides carefully and do the exercise sheets
- J. Pan et. al. Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering http://www.w3.org/2001/sw/BestPractices/SE/ODA/
- Alexander Christoph. Graph rewrite systems for software design transformations. In M. Aksit, editor, Proceedings of Net Object Days 2002, Erfurt, Germany, October 2002. Springer LNCS 2591
- D. Calvanese, M. Lenzerini, D. Nardi. Description Logics for Data Modeling. In J. Chomicki, G. Saale. Logics for Databases and Information Systems. Kluwer, 1998.
- D. Berardi, D. Calvanese, G. de Giacomo. Reasoning on UML class diagrams. Artificial Intelligence 168 (2005), pp. 70-118. Elsevier.
- Michael Kifer. Rules and Ontologies in F-Logic. Reasoning Web Summer School 2005. Lecture Notes in Computer Science, LNCS 3564, Springer. http://dx.doi.org/10.1007/11526988_2
- Mira Balaban, Michael Kifer. An Overview of F-OML: An F-Logic Based Object Modeling Language. Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010). ECEASST 2010, 36, http://journal.ub.tu-berlin.de/eceasst/article/view/537/535
- Holger Knublauch, Daniel Oberle, Phil Tetlow, Evan Wallace (ed.). A Semantic Web Primer for Object-Oriented Software Developers http://www.w3.org/2001/sw/BestPractices/SE/ODSD/
- Yi, Kwangkeun, Whaley, John, Avots, Dzintars, Carbin, Michael, Lam, Monica. Using Datalog with Binary Decision Diagrams for Program Analysis. In: Programming Languages and Systems. Lecture Notes in Computer Science 3780, 2005, pp. 97-118 http://dx.doi.org/10.1007/11575467_8
- Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Baltimore, Maryland, June 13 - 15, 2005). PODS '05. ACM, New York, NY, 1-12. DOI= http://doi.acm.org/10.1145/1065167.1065169
- Thomas, Dave, Hajiyev, Elnar, Verbaere, Mathieu, de Moor, Oege. codeQuest: Scalable Source Code Queries with Datalog, ECOOP 2006 – Object-Oriented Programming, Lecture Notes in Computer Science 4067, 2006, Springer, pp. 2 - 27 http://dx.doi.org/10.1007/11785477_2

References

- S. Ceri, G. Gottlob, L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Transactions on Knowledge And Data Engineering. March 1989, (1) 1, pp. 146-166.
- S. Ceri, G. Gottlob, L. Tanca. Logic Programming and Databases. Springer, 1989.
- Ullman, J. D. Principles of Database and Knowledge Base Systems. Computer Science Press 1989.
- Benjamin Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logics. In Proc. of World Wide Web Conference (WWW) 2003, Budapest, Hungary, 05 2003. ACM Press.
- Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. Ontologies, Meta-Models, and the Model-Driven Paradigm. Handbook of Ontologies in Software Engineering. Springer, 2006.
- http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/personen/juergen-ebert/juergen-ebert/
- Ebert, Jürgen; Riediger, Volker; Schwarz, Hannes; Bildhauer, Daniel (2008): Using the TGraph Approach for Model Fact Repositories. In: Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008). S. 9--18.
- Bildhauer, Daniel; Ebert, Jürgen (2008): Querying Software Abstraction Graphs. In: Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), collocated with ICPC 2008.

Graph Rewriting

- Graph rewriting for programs and models:
  - U. Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, 5th Int. Workshop on Graph Grammars and Their Application To Computer Science, volume 1073 of Lecture Notes in Computer Science, pages 321-335. Springer, Heidelberg, November 1994.
  - Uwe Aßmann. How to uniformly specify program analysis and transformation. In P. A. Fritzson, editor, Proceedings of the International Conference on Compiler Construction (CC), volume 1060 of Lecture Notes in Computer Science, pages 121-135. Springer, Heidelberg, 1996.
  - U. Aßmann. Graph Rewrite Systems for Program Optimization. ACM Transactions on Programming Languages and Systems, June 2000.
  - U. Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. Graph Grammar Handbook, Vol. II, 1999. Chapman&Hall.
  - U. Aßmann. Reuse in Semantic Applications. REWERSE Summer School. July 2005. Malta. Reasoning Web, First International Summer School 2005, number 3564 in Lecture Notes in Computer Science. Springer.
  - Alexander Christoph. GREAT - a graph rewriting transformation framework for designs. Electronic Notes in Theoretical Computer Science (ENTCS), 82(4), April 2003.

Goals

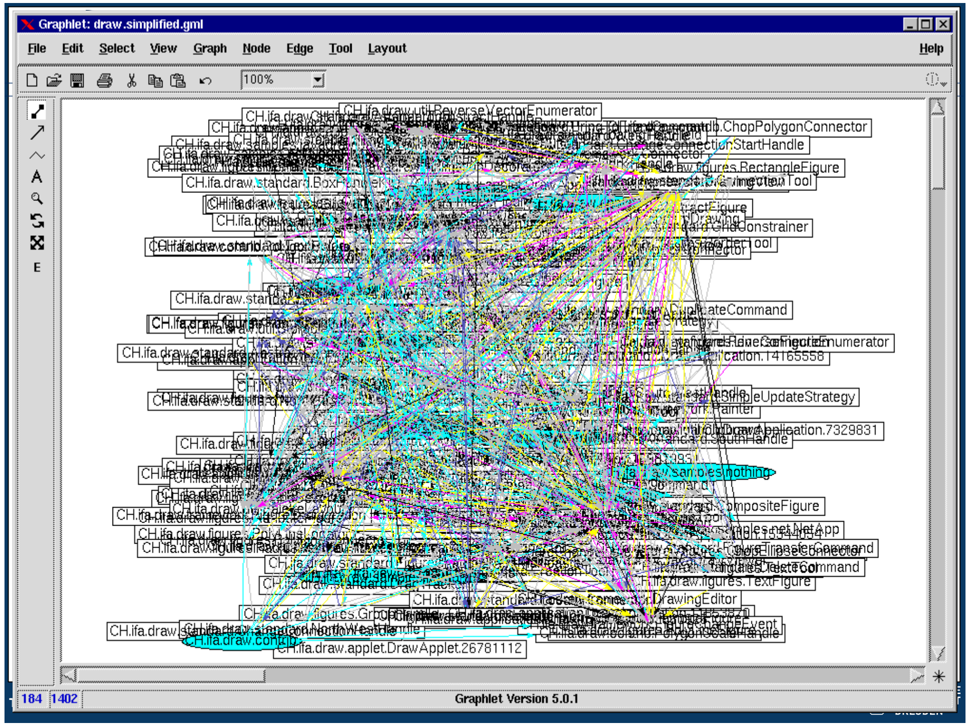
- Understand that software models can become very large
  - the need for appropriate techniques to handle large models
    - in hand development
    - automatic analysis of the models
- Learn how to use graph-based techniques to analyze and check models for consistency, well-formedness, integrity
  - Datalog, Graph Query Languages, Description Logic, EARS, graph transformations
- Understand some basic concepts of simplicity in software models

Motivation

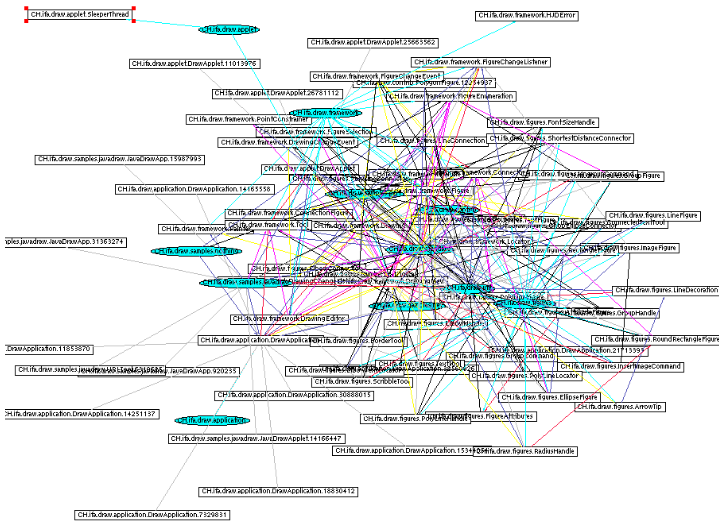
- Software engineers must be able to
  - handle *big* design specifications (design models) during development
  - work with *consistent* models
  - *measure* models and implementations
  - *validate* models and implementations
- Real models and systems become very complex
- Most specifications are graph-based
  - We have to deal with basic graph theory to be able to measure well
- Every analysis method is very welcome
- Every structuring method is very welcome

- Large models have large graphs
- They can be hard to understand
- Figures taken from Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe]

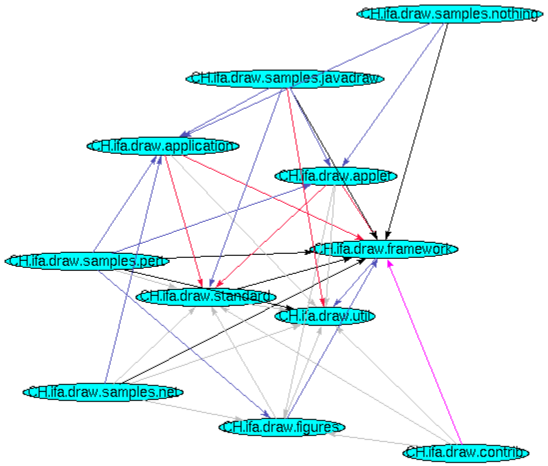
12.1 THE PROBLEM: HOW TO MASTER LARGE MODELS



Partially Collapsed



Totally Collapsed



Requirements for Modeling in Requirements and Design

- We need guidelines how to develop simple models
- We need analysis techniques to
  - Analyze models
    - Find out about their complexity
    - Find out about simplifications
  - Search in models
  - Check the consistency of the models

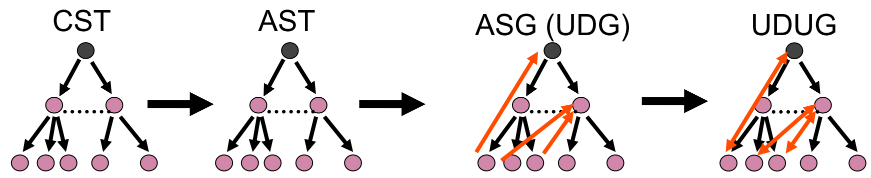
How are models and programs represented in a Software Tool?

Some Relationships (Graphs) in Software Systems

12.2 GENERATING GRAPHS FROM MODELS AND SOFTWARE

All Specifications and All Programs Have an Internal Graph-Based Representation

- Texts are parsed to abstract syntax trees (AST)
  - Two-step procedure
    - Concrete Syntax Tree (CST)
    - Abstract Syntax Tree (AST)
- Through name analysis, they become abstract syntax graphs (ASG) or Use-Def-Graphs (UDG)
- Through def-use-analysis, they become Use-def-Use Graphs (UDUG)



Concrete Syntax Tree (CST) – Example

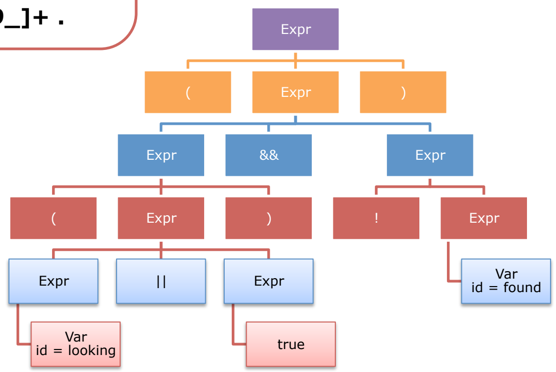
Expr ::= '(' *Expr* ')'
| Expr '&&' *Expr*
| Expr '||' expr
| '!' *Expr*
| Lit .
Lit ::= Var | '*true*' | '*false*'.
Var ::= [a-z][a-z 0-9_]+ .

Parsing this string:
((looking || true) && !found)

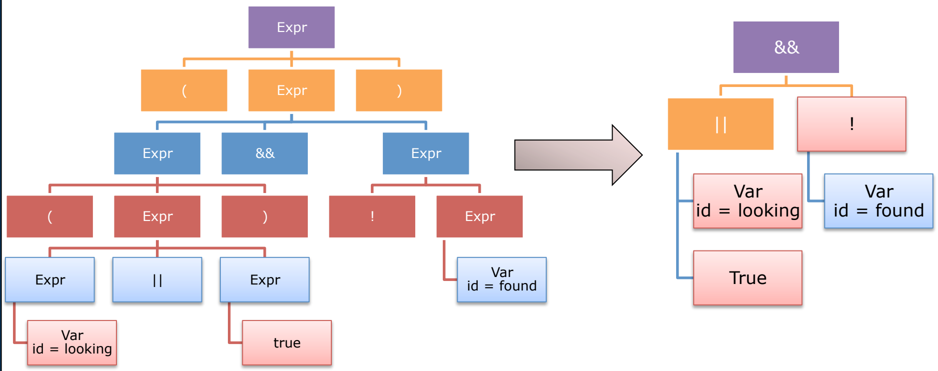
CST - Example

Expr ::= '(' *Expr* ')'
| Expr '&&' *Expr*
| Expr '||' expr
| '!' *Expr*
| Lit .
Lit ::= Var | '*true*' | '*false*'.
Var ::= [a-z][a-z 0-9_]+ .

Parsing this string:
((looking || true) && !found)

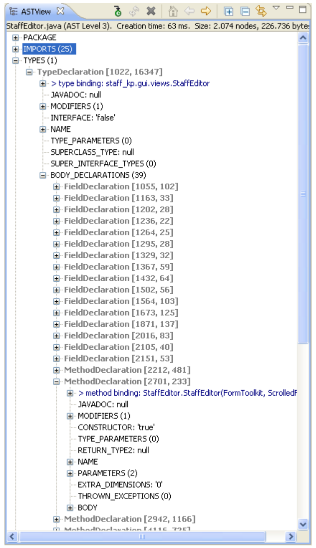


From the CST to the AST



Abstract Syntax Trees (AST)

- Parse trees (CST) waste a fair amount of space for representation of terminal symbols and productions
- Compilers post-process parse trees into ASTs
- ASTs are the fundamental data structure of IDEs (ASTView in Eclipse JDT)



AST

- Problem with ASTs: They do not support static semantic checks, re-factoring and browsing operations, e.g:
  - Name semantics:
    - Have all used variables been declared? Are they declared once?
    - Have all Classes used been imported?
  - Are the types used in expressions / assignments compatible? (type checking)
  - Referencing:
    - Navigate to the declaration of method call / variable reference / type
  - How can I pretty-print the AST to a CST again, so that the CST looks like the original CST
    - Necessary for *hygenic refactoring*

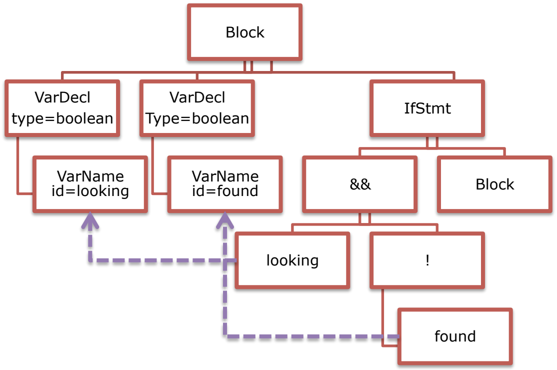
Def-Use Graphs (DUG) and Use-Definition-Use Graphs (UDUG)

- Every language and notation has
  - *Definitions* of items (definition of the variable Foo), who add type or other metadata
  - *Uses* of items (references to Foo)
- We talk in specifications or programs about *names of objects* and their use
  - Definitions are done in a data definition language (DDL)
  - Uses are part of a data query language (DQL) or data manipulation language (DML)
- Starting from the abstract syntax tree, *name analysis* finds out about the definitions of uses of names
  - Building the *Use-Def graph*
  - This revolves the meaning of used names to definitions
  - Inverting the Use-Def graph to a Use-Def-Use graph (UDUG)
  - This links all definitions to their uses

Abstract Syntax Graphs (ASG) are UDGs

- Abstract Syntax Graphs have *use-def edges* that reflect semantic relationships
  - from uses of names to definitions of names
- These edges are used for static semantic checks
  - Type checking
  - Casts and coercions
  - Type inference

```
boolean looking, found;
...
if (looking && !found ) {...}
```



Refactoring on Complete Name-Resolved Graphs (Use-Def-Use Graphs)

- UDUGs are used in refactoring operations (e.g. renaming a class or a method consistently over the entire program).
- For renaming of a definition, all uses have to be changed, too
  - We need to trace all uses of a definition in the Use-Def-graph, resulting in its inverse, the *Def-Use-graph*
  - Refactoring works always on Def-Use-graphs *and* Use-Def-graphs, the *complete name-resolved graph* (the *Use-Def-Use graphs*)

Example: Rename Refactorings in Programs

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

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

Definition

Reference (Use)

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


Refactoring

- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions
- Refactoring can be supported by tools
  - The Use-Def-Use-graph forms the basis of refactoring tools
- However, building the Use-Def-Use-Graph for a complete program costs a lot of space and is a difficult program analysis task
  - Every method that structures this graph benefits immediately the refactoring
  - either simplifying or accelerating it
- UDUGs are large
  - Efficient representation important

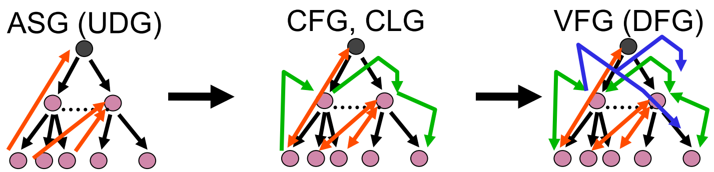
Further Representations

From the ASG or an UDUG, more graph-based program representations can be derived

- Control-flow Analysis -> Control-Flow Graph (CFG), Call graph (CLG)
  - Records control-flow relationships
- Data-Flow Analysis -> Data-Flow Graph (DFG) or Value-Flow Graph (VFG)
  - Records flow relationships for data values

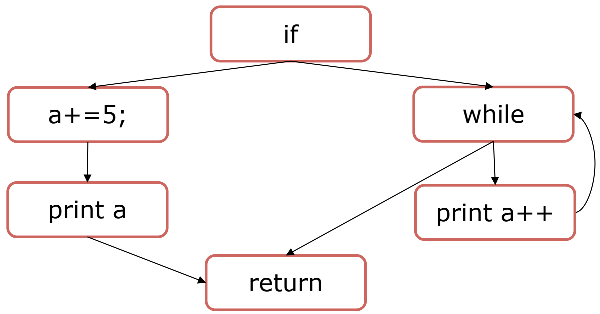
The same remarks holds for graphic specifications

- Hence, all specifications are graph-based!



Control-Flow Graphs

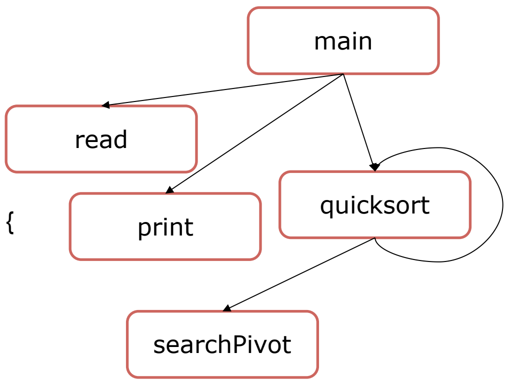
- Describe the control flow in a program
- Typically, if statements and switch statements split control flow
  - Their ends join control flow
- Control-Flow Graphs *resolve* symbolic labels
  - Perform name analysis on labels
- Nested loops are described by nested control flow graphs



Simple (Flow-Insensitive) Call Graph (CLG)

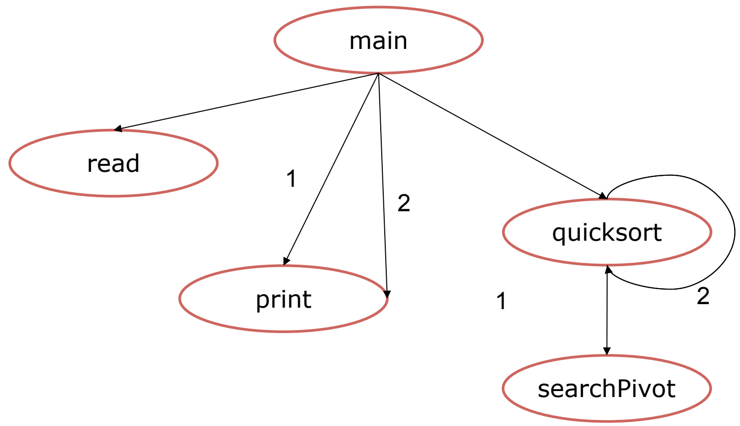
- Describe the call relationship between the procedures
  - Interprocedural control-flow analysis performs name analysis on called procedure names

```
main = procedure () {
  array int[] a = read();
  print(a);
  quicksort(a);
  print(a);
}
quicksort = procedure(a: array[0..n]) {
  int pivot = searchPivot(a);
  quicksort(a[0], a[pivot-1]);
  quicksort(a[pivot+1,n]);
}
```



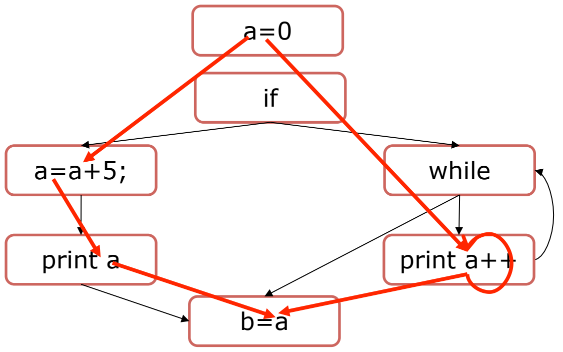
(Flow-Insensitive) Call Graph (CLG)

- Describe the call relationship between the procedures including call sites
  - Flow-insensitive
  - Flow-sensitive versions consider the control flow graph



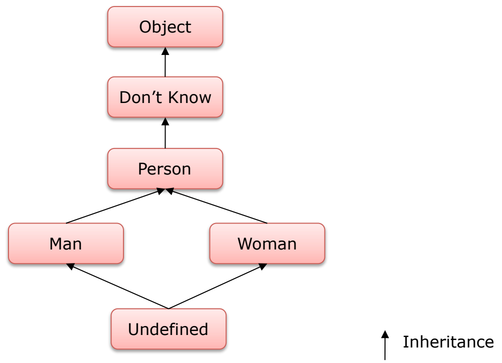
Data-Flow Graphs (DFG)

- A *data-flow graph (DFG)* aka *value-flow graph (VFG)* describes the flow of data through the variables
  - DFG are based on control-flow graphs
- Building the data-flow graph is called *data-flow analysis*
  - Data-flow analysis is often done by *abstract interpretation,* the symbolic execution of a program at compile time



Inheritance Analysis: Building an Inheritance Tree or Inheritance Lattice

- A *lattice* is a partial order with largest and smallest element
- Inheritance hierarchies can be generalized to inheritance lattices
- An *inheritance analysis* builds the transitive closure of the inheritance lattice

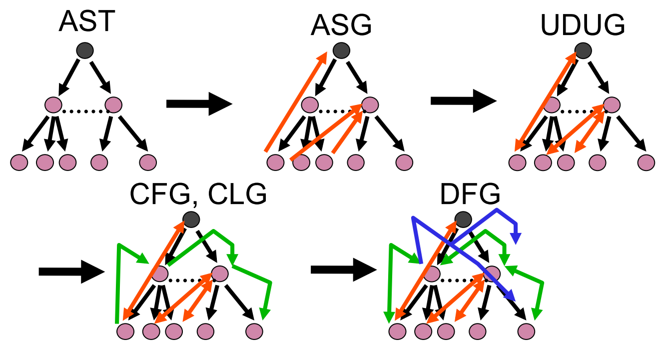


UML Graphs

- All diagram sublanguages of UML generate internal graph representations
  - They can be analyzed and checked with graph techniques
  - Graphic languages, such as UML, need a graph parser to be recognized, or a specific GUI who knows about graphic elements
- Hence, graph techniques are an essential tool of the software engineer

Remark: All Specifications Have a Graph-Based Representation

- Texts are parsed to abstract syntax trees (AST)
- Graphics are parsed by GUI or graph parser to AST also
- Through name analysis, they become abstract syntax graphs (ASG)
- Through def-use-analysis, they become Use-def-Use Graphs (UDUG)
- Control-flow Analysis -> CFG, CLG
- Data-Flow Analysis -> DFG



Lists, Trees, Dags, Graphs

Structural constrains on graphs

(background information)

12.3 TYPES OF GRAPHS IN SPECIFICATIONS

Modeling Graphs on Two Abstraction Levels

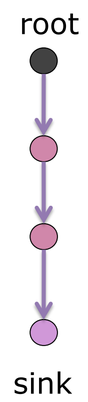
- In modeling, we deal mostly with *directed graphs (digraphs)* representing unidirectional relations
  - lists, trees, dags, overlay graphs, reducible (di-)graphs, graphs
- There are two different abstraction levels; we are interested in the logical level:
  - **Logical level** (conceptual, abstract, often declarative, problem oriented)
    - Methods to specify graph and algorithms on graphs:
      - Relational algebra
      - Datalog, description logic
      - Graph rewrite systems, graph grammars
      - Recursion schemas
  - **Physical level** (implementation level concrete, often imperative, machine oriented)
    - Representations: Data type adjacency list, boolean (bit)matrix, BDD
    - Imperative algorithms
    - Pointer based representations and algorithms

Essential Graph Definitions

- Fan-in
  - In-degree of node under a certain relation
  - Fan-in(n = 0): n is *root* node (*source*)
  - Fan-in(n) > 0: n is *reachable* from other nodes
- Fan-out
  - Out-degree of node under a certain relation
  - Fan-out(n) = 0: n is *leaf* node (*sink*)
  - An *inner node* is neither a root nor a leaf
- Path
  - A path p = (n_1, n_2,…,n_k) is a sequence of nodes of length k

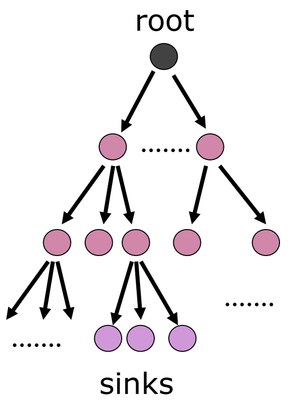
Lists

- One source (root)
- One sink
- Every other node has fan-in 1, fan-out 1
- Represents a *total order* (sequentialization)
- Gives
  - Prioritization
  - Execution order



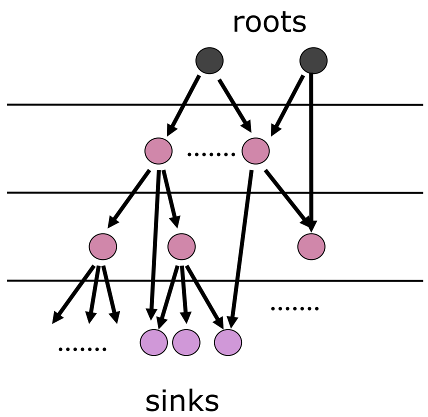
Trees

- One source (root)
- Many sinks (leaves)
- Every node has fan-in <= 1
- *Hierarchical abstraction*:
  - A node *represents* or *abstracts* all nodes of a sub tree
- Example
  - SA function trees
  - Organization trees (line organization)



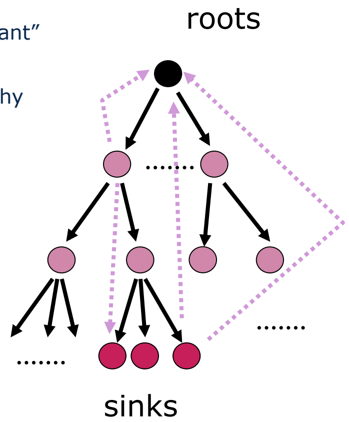
Directed Acyclic Graphs

- Many sources
  - A jungle (term graph) is a dag with one root
- Many sinks
- Fan-in, fan-out arbitrary
- Represents a partial order
  - Less constraints that in a total order
- Weaker hierarchical abstraction feature
  - Can be layered
- Example
  - UML inheritance dags
  - Inheritance lattices



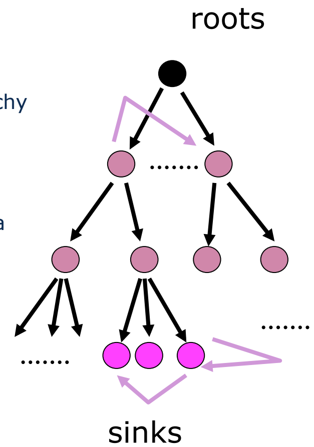
Skeleton Trees with Overlay Graphs (Trees with Secondary Graphs)

- Skeleton tree with **overlay graph** (secondary links)
  - Skeleton tree is primary
  - Overlay graph is secondary: "less important"
- Advantage of an Overlay Graph
  - Tree can be used as a conceptual hierarchy
  - References to other parts are possible
- Example
  - XML, e.g., XHTML. Structure is described by Xschema/DTD, links form the secondary relations
  - AST with name relationships after name analysis (name-resolved trees, abstract syntax graphs)

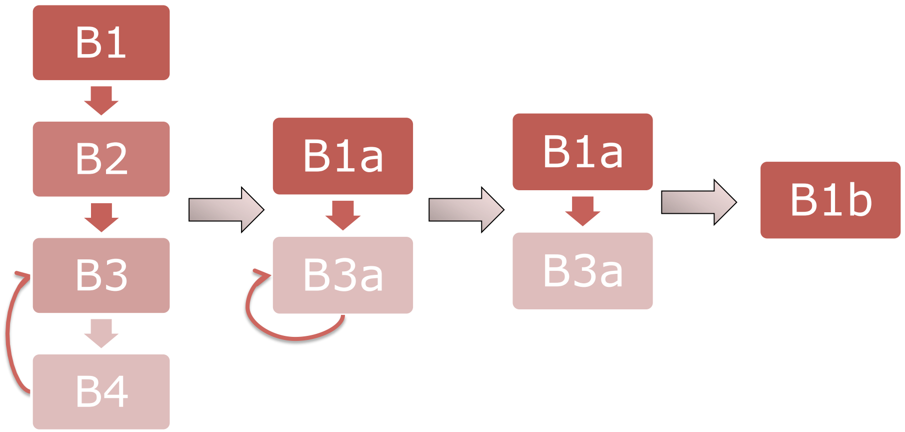


Reducible Graphs (Graphs with Skeleton Trees)

- A **reducible graph** is a graph with cycles, however, only between siblings
  - No cycles between hierarchy levels
- Graph can be "reduced" to one node
- Advantage
  - Tree can be used as a conceptual hierarchy
- Example
  - UML statecharts
  - UML and SysML component diagrams
  - Control-flow graphs of Modula, Ada, Java (not C, C++)
  - SA data flow diagrams
  - Refined Petri Nets

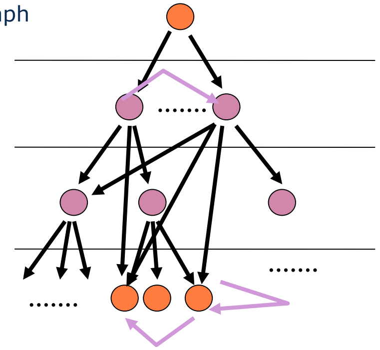


Reduction of a Reducible Graph



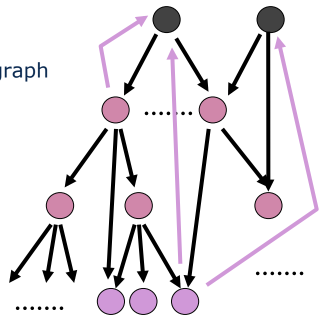
Layerable Graphs with Skeleton Dags

- Like reducible graphs, however, sharing between different parts of the skeleton trees
  - Graph cannot be "reduced" to one node
- Advantage
  - Skeleton can be used to layer the graph
  - Cycles only within one layer
- Example
  - Layered system architectures

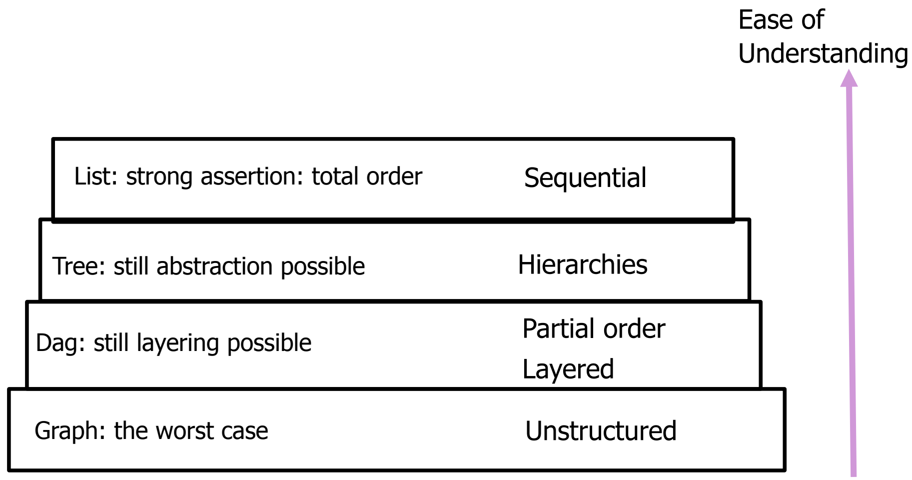


Wild Unstructured (Directed) Graphs

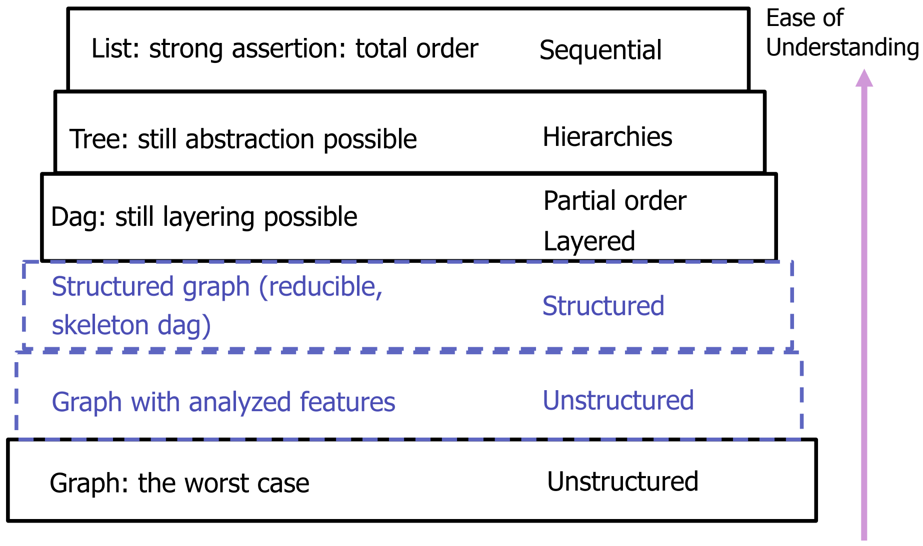
- Wild, unstructured graphs are the worst structure we can get
  - Wild, unstructured, irreducible cycles
  - Unlayerable, no abstraction possible
  - No overview possible
- Many roots
  - A digraph with one source is called flow graph
- Many sinks
- Example
  - Many diagrammatic methods in Software Engineering
  - UML class diagrams



Strength of Assertions in Models

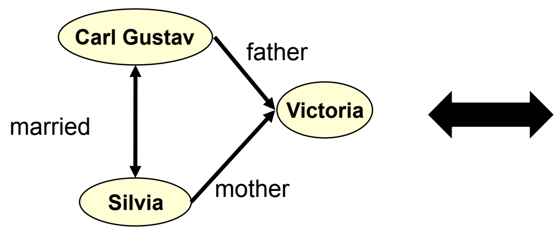


- Saying that a relation is
  - A list: very strong assertion, total order!
  - A tree: still a strong assertion: hierarchies possible, easy to think
  - A dag: still layering possible, still a partial order
  - A layerable graph: still layering possible, but no partial order
  - A reducible graph: graph with a skeleton tree
  - A graph: hopefully, some structuring or analysis is possible. Otherwise, it's the worst case
- And those propositions hold for every kind of diagram in Software Engineering!
- Try to model reducible graphs, dags, trees, or lists in your specifications, models, and designs
  - Systems will be easier, more efficient



12.4 METHODS AND TOOLS FOR ANALYSIS OF GRAPH-BASED MODELS

- In the following, we will make use of the graph-logic isomorphism:
- Graphs can be used to represent logic
  - Nodes correspond to constants
  - (Directed) edges correspond to binary predicates oder nodes (*triple statements)*
  - Hyperedges (n-edges) correspond to n-ary predicates
- Consequence:
  - Graph algorithms can be used to test logic queries on graph-based specifications
  - Graph rewrite systems can be used for deduction

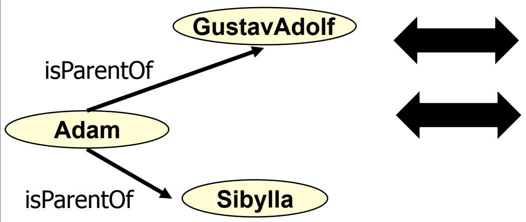


married(CarlGustav,Silvia).
married(Silvia, CarlGustav).
father(CarlGustav,Victoria).
mother(Silvia,Victoria).

CarlGustav is married to Silvia.
Silvia is married to CarlGustav.
CarlGustav is father to Victoria.
Silvia is mother to Victoria.

Graphs and Fact Data Bases

- Graphs can also be noted textually
- Graphs consist of nodes, relations
- Relations link nodes



- Fact data bases consist of constants (data) and predicates
- Nodes of graphs can be regarded as constants, edges as predicates between constants (*facts*):

```
// Triples
Adam isParentOf GustavAdolf.
Adam isParentOf Sibylla.

// Facts
isParentOf(Adam,GustavAdolf).
isParentOf(Adam,Sibylla).
```


Queries on Graph-Based Models Make Implicit Knowledge Explicit

- Since graph-based models are a mess, we try to analyze them
- Knowledge is either
  - **Explicit**, I.e., represented in the model as edges and nodes
  - **Implicit**, I.e., hidden, not directly represented, and must be analyzed
- Query and analysis problems try to *make implicit knowledge explicit*
  - E.g. Does the graph have one root? How many leaves do we have? Is this subgraph a tree? Can I reach that node from this node?
- Determining features of nodes and edges
  - Finding certain nodes, or patterns
- Determining global features of the model
  - Finding paths between two nodes (e.g., connected, reachable)
  - Finding paths that satisfy additional constraints
  - Finding subgraphs that satisfy additional constraints

Queries for Checking Consistency (Model Validation)

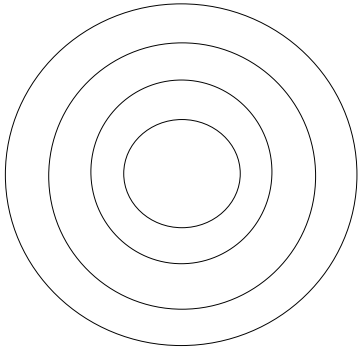
- Queries can be used to find out whether a graph is *consistent (i.e., valid, well-formed)*
  - Due to the graph-logic isomorphism, constraint specifications can be phrased in logic and applied to graphs
  - Business people call these constraint specifications *business rules*
- Example:
  - if a person hasn't died yet, its town should not list her in the list of dead people
  - if a car is exported to England, steering wheel and pedals should be on the right side; otherwise on the left

12.4.1 Layering Graphs: How to Analyze a System for Layers

- With the Same Generation Problem
- How to query a dag and search in a dag
- How to layer a dag – a simple structuring problem

Layering of Systems

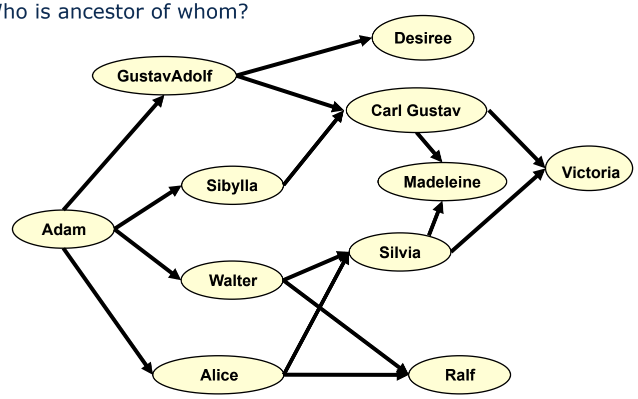
- To be comprehensible, a system should be structured in layers
  - Several relations in a system can be used to structure it, e.g., the
    - Call graph: layered call graph
    - Layered definition-use graph



- A *layered architecture* is the dominating style for large systems
- Outer, upper layers use inner, lower layers (layered USES relationship)
- Legacy systems can be analyzed for layering, and if they do not have a layered architecture, their structure can be improved towards this principle

Layering of Acyclic Graphs

- Given any acyclic relation, it can be made layered
  - SameGeneration analysis layers in trees or dags
- Example: layering a family tree:
  - Who is whose contemporary?
  - Who is ancestor of whom?



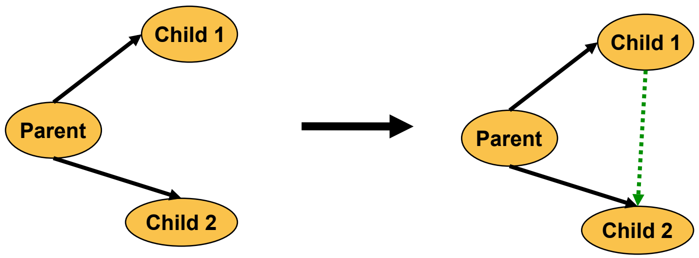
Pattern and Rules

- Parenthood can be described by a *graph pattern*
- We can write the graph pattern also in logic:

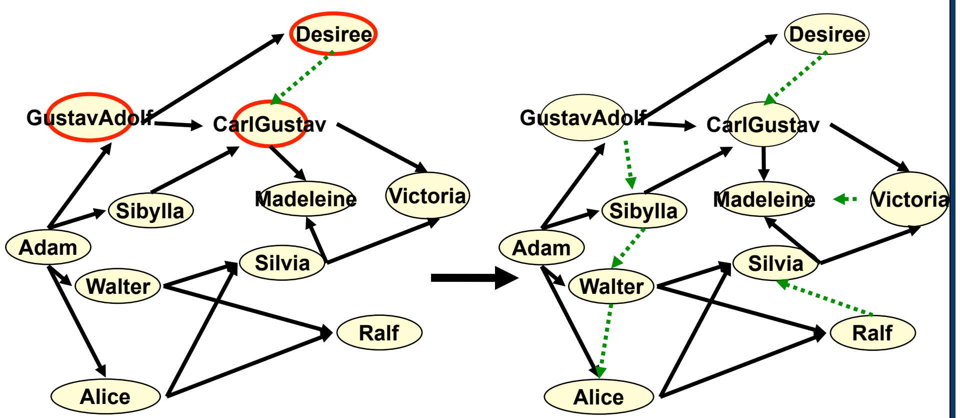
```
isParentOf(Parent,Child1) && isParentOf(Parent,Child2)
```

- And define the rule

```
if isParentOf(Parent,Child1) && isParentOf(Parent,Child2)
then sameGeneration(Child1,Child2)
```

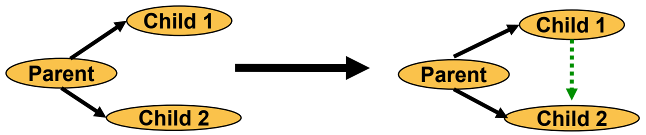


Impact of Rule on Family Graph

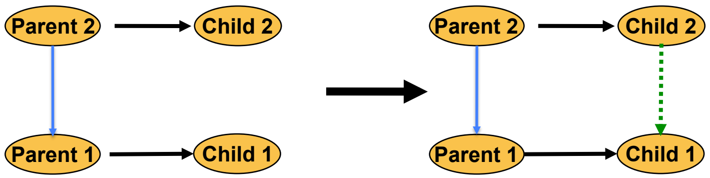


Rule set "Same Generation"

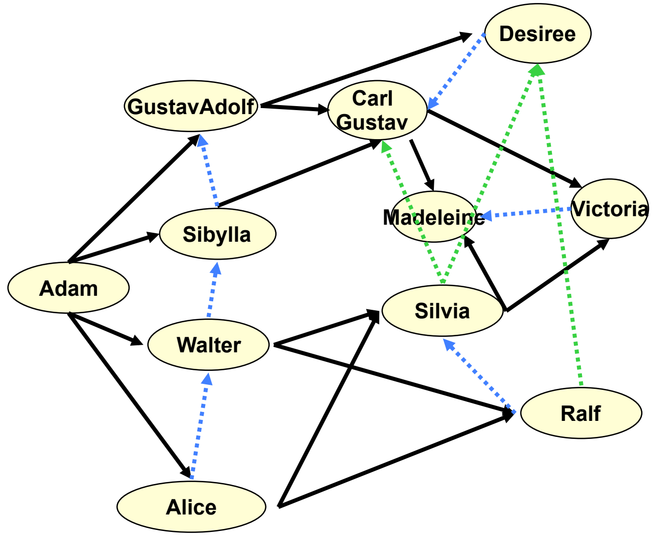
- Base rule: Beyond sisters and brothers we can link all people of same generation



- Additional rule (transitive): Enters new levels into the graph



Impact of Transitive Rule



"Same Generation" Introduces Layers

- Computes all nodes that belong to one layer of a dag
  - If backedges are neglected, also for an arbitrary graph
- Algorithm:
  - Compute Same Generation
  - Go through all layers and number them
- Applications:
  - Compute layers in a call graph
    - Find out the call depth of a procedure from the main procedure
  - Restructuring of legacy software (refactoring)
    - Compute layers of systems by analyzing the USES relationships (ST-I)
    - Insert facade classes for each layer (Facade design pattern)
      - Every call into the layer must go through the facade
    - As a result, the application is much more structured

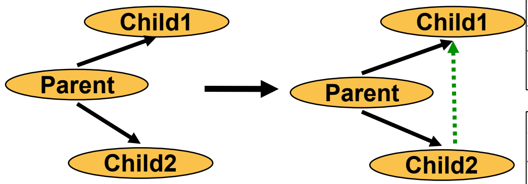
12.4.2 SEARCHING GRAPHS – SEARCHING IN SPECIFICATIONS WITH DATALOG AND EARS

SameGeneration as a Graph Rewrite System

- The rule system SameGeneration only adds edges.
- An *edge addition rewrite system (EARS)* adds edges to graphs
  - It enlarges the graph, but the new edges can be marked such that they are not put permanently into the graph
  - EARS are declarative
    - No specification of control flow and an abstract representation
    - Confluence: The result is independent of the order in which rules are applied
    - Recursion: The system is recursive, since relation sameGeneration is used and defined
    - Termination: terminates, if all possible edges are added, latest, when graph is complete
- EARS compute with graph query and graph analysis
  - Reachability of nodes
  - Paths in graphs
  - SameGeneration can be used for graph analysis

Rule Systems in EARS and Datalog

- Rule systems can be noted textually or graphically (DATALOG or EARS)
- Datalog contains
  - textual if-then rules, which test predicates about the constants
  - rules contain variables




```
// conclusion
sameGeneration(Child1, Child2)
:-   // say: "if"
// premise
isParentOf(Parent,Child1),
isParentOf(Parent,Child2).
```

```
// premise
if isParentOf(Parent,Child1) &&
isParentOf(Parent,Child2)
then
// conclusion
sameGeneration(Child1,Child2)
```


Same Generation Datalog Program

isParentOf(Adam,GustavAdolf).
isParentOf(Adam,Sibylla).
.....
if isParentOf(Parent,Child1), isParentOf(Parent,Child2)
then sameGeneration(Child1, Child2).
if sameGeneration(Parent1,Parent2),
isParentOf(Parent1,Child1), isParentOf(Parent2,Child2)
then
sameGeneration(Child1, Child2).

Searching and Solving Path Problems is Easy With Datalog

 # A SMPP problem (searching for Single source a set of Multiple targets)
descendant(Adam,X)?
X={ Silvia, Carl-Gustav, Victoria, ...}

 # An MSPP problem (multiple source, single target)
descendant(X,Silvia)?
X={Walter, Adam, Alice}

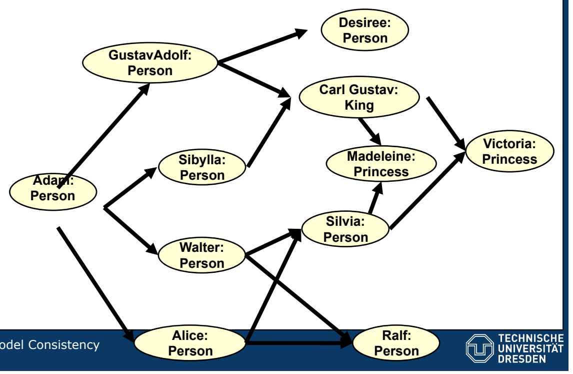
 # An MMPP problem (multiple source, multiple target)
ancestor(X,Y)?
{X=Walter, Y={Adam}
X=Victoria, Y={CarlGustav, Silvia, Sibylla, ...}

Y = Adam, Walter, ...
 # Victoria, Madeleine, CarlPhilipp not in the set

F-Datalog and OWL (Description Logic, DL)

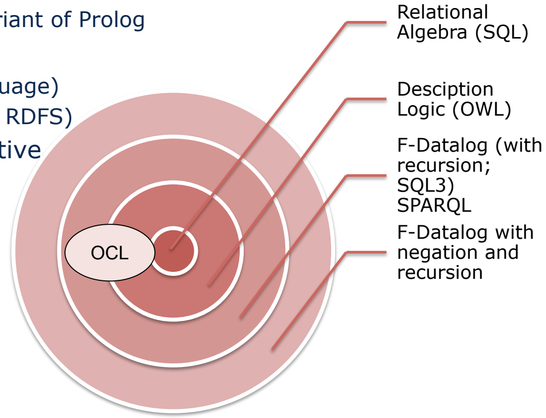
- F-Datalog and DL are special forms of *typed binary* Datalog (typed EARS)
  - Only with unary (classes) and binary relations (relationships)
  - Classes and objects as types, relationship types and relations
  - Inheritance of classes and relationships
  - Frame-based (like UML-CD)
  - F-Datalog has Closed-World Assumption (CWA), i.e., treats incomplete information as FALSE
- OWL (Web Ontology Language):
  - Triple-, not frame-based - all knowledge is specified with *triples*
  - OWL has a cleanly defined sublanguage hierarchy
  - OWL has Open-World Assumption (OWA), i.e., treats incomplete information as TRUE

```
Adam instanceOf Person.
Sibylla instanceOf Person.
GustavAdolf instanceOf Person.
King isA Person.
Princess isA Person.
...
Adam parentOf GustavAdolf.
Adam parentOf Sibylla.
...
```

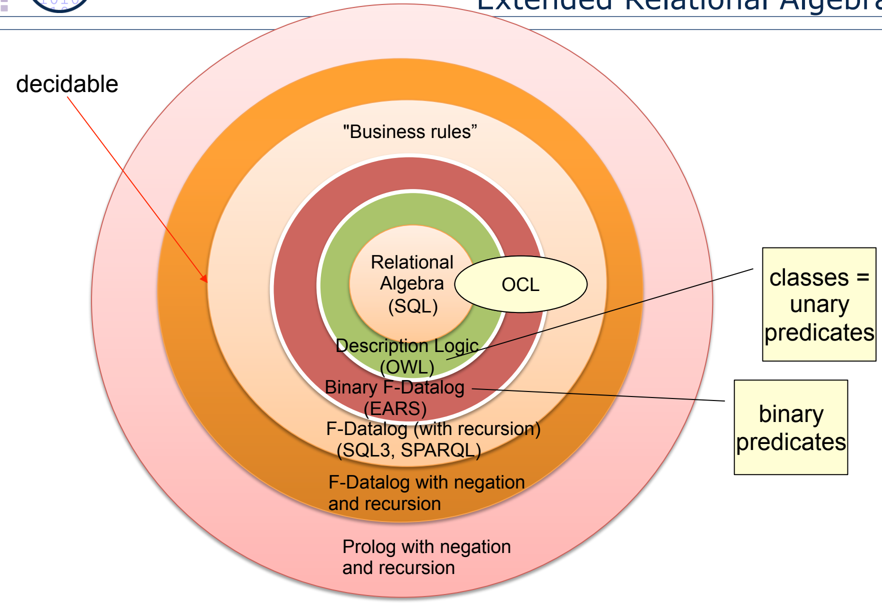


F-Datalog, DL, OCL, and EARS: Extended Relational Algebra

- F-Datalog, DL and EARS correspond to relational Algebra with recursion (see lecture on data bases).
  - SQL has no recursion, SQL-3 has
  - Negation can be added
  - F-Datalog is a simple variant of Prolog
- DL languages:
  - OWL (ontology web language)
  - SPARQL (SQL like QL for RDFS)
- OCL does not have transitive closure, but iteration



Datalog, DL, OCL, and EARS: Extended Relational Algebra



Application Areas of Datalog, DL, OCL, and EARS

- See the new language F-OML [Baladan/Kifer], in which UML-CD are interpreted as graphs and queried with F-Datalog
- Graph query problems (searching graphs)
  - Reachability of nodes (transitive closure, SSPP, etc.)
- Consistency checking of graph-based specifications
  - Name analysis (building def-use graphs)
  - Data analysis
  - Program analysis
    - Building control-flow graphs
    - Value-flow analysis
  - Model analysis (UML, OWL)
- Structurings and algorithms on structured graphs
  - Layering of system relations
  - Reducibility
  - Strongly connected components
- Specification of contracts for procedures and services
  - Prover can statically prove the validity of the contract

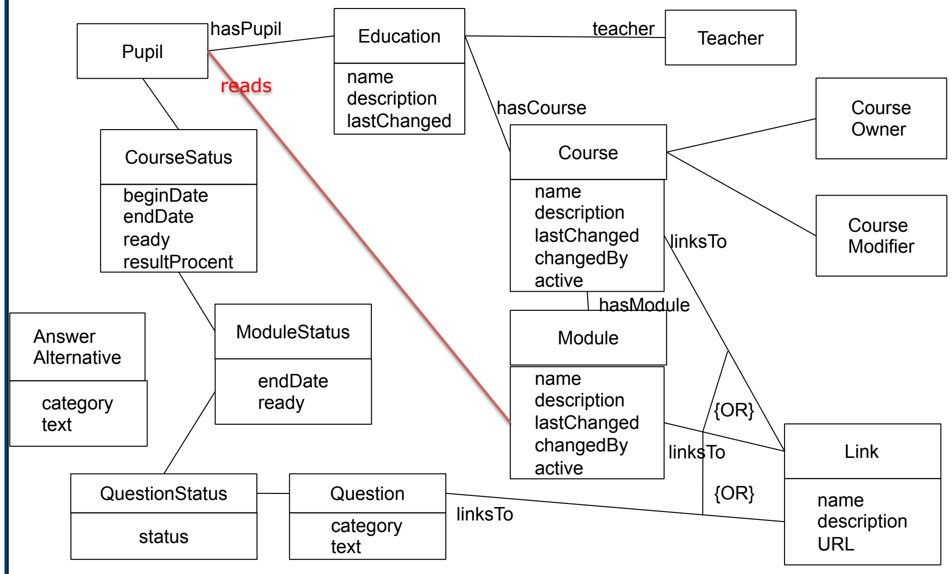
➢Step 1: encode the diagram into a Datalog or DL fact base

➢Step 2: define integrity constraint rules

➢Step 3: let the rules run

12.4.3 EXAMPLE FOR MODEL VALIDATION: CHECKING UML DIAGRAMS WITH F-DATALOG

Example: The Domain Model of the Web-Based Course System



Searching with F-Datalog or DLQueries on UML Class Diagrams

```
// Step 1: construct fact base: the UML class diagram
// in Datalog fact syntax.
// Object declarations:
programming:Education. john:Person. mary:Person. lisp:Module
// Edge fact declarations:
teacher(programming,john).
hasCourse(programming, lisp).
hasPupil(programming,mary).
hasModule(lisp,closures).

// Step 2: construct integrity constraint rules
reads(Person,Module) :-
   hasPupil(Person,E), hasCourse(E,C), hasModule(C,Module).

// Step 3: let rules run: form and execute a query
:- reads(mary, Module)
// the answer
>> Module = closures
```


Example: Web Queries with Logic

- ➢ The Web is a gigantic graph
  - ➢ Pages are trees, but links create real graphs
  - ➢ Links are a secondary structure which overlays the primary tree structure
  - ➢ Graph algorithms and queries can be applied to the web
- ➢ RDFS (resource description framework schema) is used as DDL
  - ➢ a simple graph language for triple specifications
  - ➢ classes, inheritance, inheritance on binary relations, expressions and queries on binary relations
- ➢ SPARQL as query language (triple querying with SQL-like language
- ➢ OWL adds inheritance analysis (subsumption analysis)
- ➢ Other experimental languages:
  - ➢ F-Datalog/Flora/XSB (M. Kifer, NY Stony Brook), Florijd (Freiburg)
  - ➢ OntoBroker von Ontoprise.com: http://www.ontoprise.de/deutsch/start/produkte/ontobroker/, based on F-Datalog
- ➢ New languages are being developed
  - ➢ In the European network REWERSE (www.rewerse.net)
- ➢ www.w3c.org

The Swiss-Knife of Graph Analysis

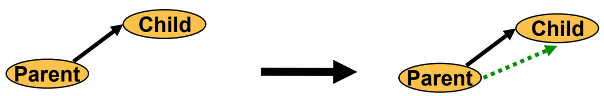
12.5 REACHABILITY QUERIES WITH TRANSITIVE CLOSURE IN F-DATALOG AND EARS

Who is Descendant of Whom?

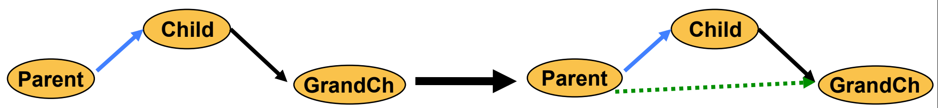
- Sometimes we need to know *transitive* edges, i.e., edges after edges of the same color
  - Question: what is *reachable* from a node?
  - Which descendants has Adam?
- Answer: Transitive closure calculates *reachability* over nodes
  - It contracts a graph, inserting masses of edges to all reachable nodes
  - It contracts all paths to single edges
  - It makes reachability information explicit
- After transitive closure, it can easily be decided whether a node is reachable or not
  - Basic premise: base relation is *not changed* (offline problem)

Transitive Closure as Datalog Rule System or EARS

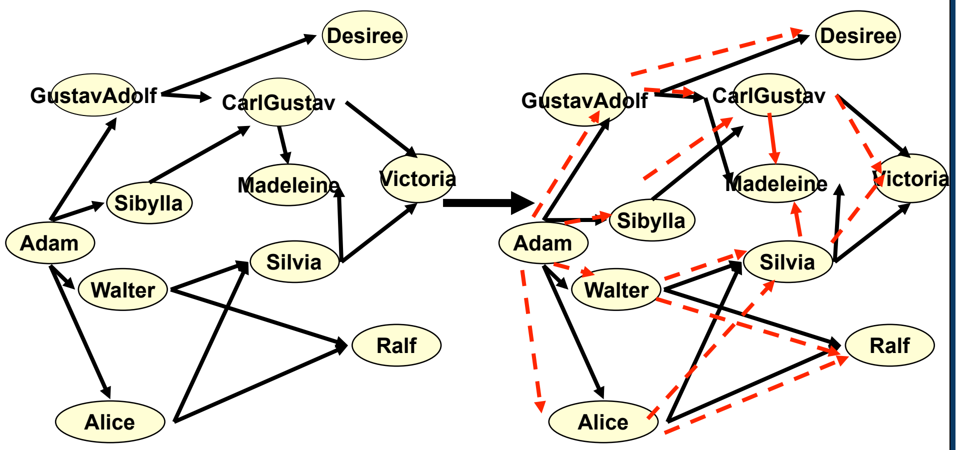
- Basic rule `descendant(V,N) :- isChildOf(V,N).`



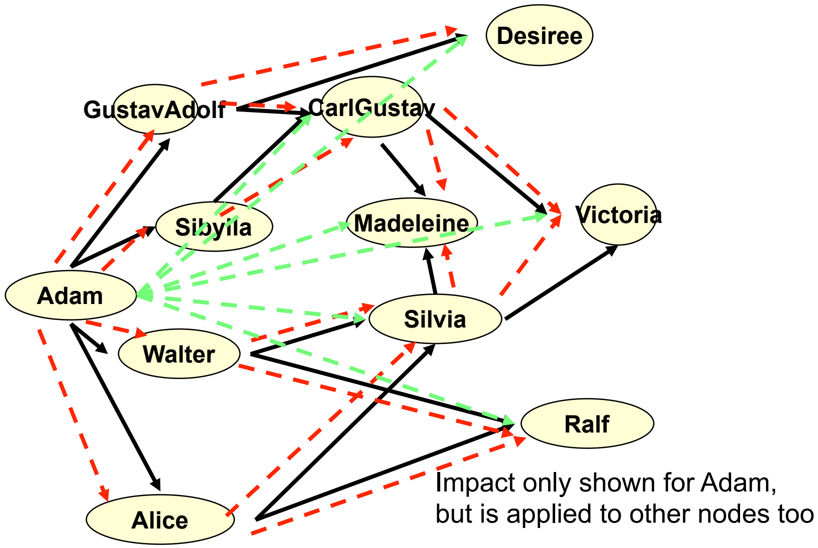
- Transitive rule (recursion rule)
- left recursive: `descendant(V,N) :- descendant(V,X),isChildOf(X,N).`
- right recursive: `descendant(V,N) :- isChildOf(V,X), descendant(X,N).`



Impact of Basic Rule



Impact of Recursion Rule



[S|M][S|M]PP Path Problems are Special Cases of Transitive Closure

- Single Source Single Target Path Problem, SSPP:
  - Test, whether there is a path from a source to a target
- Single Source Multiple Target SMPP:
  - Test, whether there is a path from a source to several targets
  - Or: find n targets, reachable from one source
- Multiple Source Single Target MSPP:
  - Test, whether a path from n sources to one target
- Multiple Source Multiple Target MMPP:
  - Test, whether a path of n sources to n targets exists
- All can be computed with transitive closure:
  - Compute transitive closure
  - Test sources and targets on direct neighborship

Exercise: Railway Routes as Reachability Queries

- Base (Facts):
  - `directlyLinked(Berlin, Potsdam).`
  - `directlyLinked(Potsdam,Braunschweig).`
  - `directlyLinked(Braunschweig, Hannover).`
- Define the predicates
  - `linked(A,B)`
  - `alsoLinked(A,B)`
  - `unreachable(A,B)`
- Answer the queries
  - `linked(Berlin,X)`
  - `unreachable(Berlin, Hannover)`

Application: Inheritance Analysis as Reachability Queries

- Base (Facts):
  - `class(Person). class(Human). class(Man). class(Woman).`
  - `extends(Person, Human).`
  - `extends(Man,Person).`
  - `extends(Woman,Person).`
- Define the predicates
  - `superScope(A,B) :- class(A), class(B), isA(A,B).`
  - `transitiveSuperScope(A,B) :- superScope(A,C), transitiveSuperScope(C,B).`
- Answer the queries
  - `? transitiveSuperScope(Man,X)`
  - `>> {X=Person,X=Human}`
  - `? transitiveSuperScope(Woman,Y)`
  - `>> {Y=Person,Y=Human}`

Operator Transitive Closure

- Transitive closure can be defined as **higher-order Operator (Skeleton)**:

```
Operator *<rel,baseRel>(A,B) = {
  rel(A,B) :- baseRel(A,B).
  rel(A,B) :- rel(A,C),baseRel(C,B).
```

- With that holds:
  - `transitiveSuperScope(A,B) :- isA*(A,B).`
- Operator **positive transitive closure**:

```
Operator +<rel,baseRel>(A,B) = {
  rel(A,B) :- baseRel(A,C), baseRel(C,B).
  rel(A,B) :- rel(A,C),baseRel(C,B).
```

- With that holds:
  - `realSuperClass(A,B) :- isA+(A,B).`

Cost of Transitive Closure

- Transitive closure (TC) has many implementations
  - Naive: multiplication of boolean matrices $O(n^3)$
  - Multiplication of boolean matrices with Russian Method is $O(n^{2.4})$
  - Nested-loop joins from relational algebra: $O(n^3)$
    - Gets better with semi-naive evaluation, hashed joins, semi-joins, and indices
  - Munro/Purdue algorithm is almost linear, but costs space

Transitive Closure and Several Relations

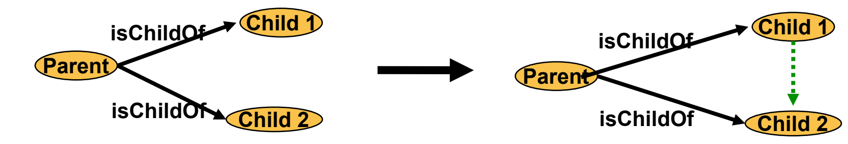
- Transitive closure can work on *several* relations
- If we want to know, whether a certain node is reachable under several relations
  - Compute transitive closure on all of them
  - Test neighbor ship directly
- This delivers an implementation of the existential quantifier for logic

Central Theorem of Datalog/DL/EARS

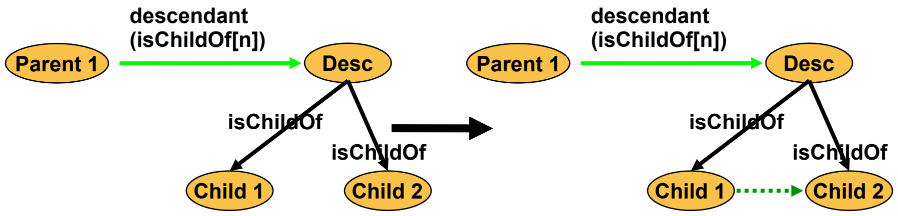
- Any Datalog program or EARS graph rewrite system can be transformed into an equivalent one without recursion
  - And only applies the operator Transitive Closure
  - (The transitive closure uses direct recursion, but encapsulates it)
- What does this mean in practice? (Remember, Datalog/EARS can be used to specify consistency constraint on graph-based specifications)

Ex.: SameGeneration as Non-Recursive System

- Basic rule as before



- Additional non-recursive rule (descendant is transitive closure of isChildOf)



Applications of Graph Reachability in Consistency Checking

- **Corollary**: To solve an arbitrary reachability problem, use a non-recursive query and the operator TransitiveClosure.
- **Consequence**: should a graph-based specification be checked on consistency (by evaluation of consistency constraints),
  - it can be done with non-recursive Datalog query and the operator TransitiveClosure
  - And solved with the complexity of a good TransitiveClosure algorithm
- **Precondition**: the input graphs are fix, i.e., do not change (static problem)
- Since the relation is one of the qualities of the world this is a central problem of computer science and IT
  - Similar to searching and sorting

Dynamic Graph Reachability and its Applications

- The Reps/Ramalinguan Checking Theorem: (1997):
  - An online analysis and constraint-checking problem is a problem that is specified by Datalog, EARS, or definite set constraints, in which the basic relations are changed online (dynamic graph reachability problem)
  - An online analysis problem can be reduced to context-sensitive graph reachability resp. dynamic transitive closure
  - and be computed in O(n3) (cubic barrier problem)
- Applies to many problems in modeling, requirement analysis, design consistency:
  - If you can reduce a consistency or structuring problem to static or dynamic graph reachability, you have almost won since Datalog and transitive closure are powerful tools

Generic Datalog Queries

- Transitive closure is a general graph operator
  - Computing reachability
  - Can be applied generically to all relations!
- Many other Datalog rule systems are also generic operators
  - sameGeneration
  - stronglyConnectedComponents
  - dominators
- And that's why we consider them here:
  - They can be applied to design graphs
  - Is class X reachable from class Y?
  - Show me the ancestors in the inheritance graph of class Y
  - Is there a cycle in this cross-referencing graph?

Ex. The Query Language TGreQL

- Prof. J. Ebert U Koblenz

```
From caller, callee: V{Method}
With caller [
   ← {isStatementIn}
  [ ← {isReturnValueOf} ]
   ← {isActualParameterOf} *
   ← {isCalleeOf}
 ] +
Report
  caller.name as „Caller“
  callee.name as „Callee“
```

- * Transitive closure operator
- + positive transitive closure
- ← navigation direction
- [] optional path
- ( ) sequence of paths or edges
- | alternative path

Caller	Callee
main	System.out.println
main	compute
main	twice
main	add
compute	twice
compute	add

When a specification becomes big...

12.6 APPLICATION: CONSISTENCY CHECKING OF GRAPH-BASED MODELS

Example 1: Consistency Checking for Car Specifications

- Car data specifications in the MOST standard
  - Thousands of parts, described for an entire supplier industry
  - Many inconsistencies possible
  - Due to human errors
- Global variants of the cars must be described
- Examples of context conditions for global variants of cars:
  - The problem of English cars: A steering wheel on the right implies accelerator, brake, clutch on the right
  - Automatic gears: an automatic gear box requires an automatic gear-shift lever

First Idea

- Define a context free grammar for the car data
- From that, derive a XML schema for the car data
  - Enrich the grammar nonterminals with attributes
- Parse the data and validate it according to its context free structure

Second Idea

- Analyze consistency of the specifications by regarding them as graphs
- Check definition criterion (name analysis)
  - “is every name I refer to defined elsewhere”?
- Analyze layers with SameGeneration
  - How many layers does my car specification have?
  - Is it acyclic?
- Write a query that checks the consistency global variants
  - If the car is to be exported to England, the steering wheel, the pedals should be on the right side
  - If the car has an automatic gear box, it must have an automatic gear-shift lever

Third Idea: Use Logic Language

- OWL (description logic) can be used for consistency constraints, also of car specifications
  - Result: an *ontology*, a vocabulary of classes with consistency constraints
  - OWL engines (RACER, Triple) can evaluate the consistency of car specifications
  - Ontologies can formulate consistency criteria for an entire supplier chain [Aßmann2005]
- Typed (F-Datalog) can be used for recursive consistency constraints
  - Ontoprise reasoner
  - XSB F-Datalog plugin

Example 2: Consistency Checking of Tax Declarations

- Task: you have been hired by the tax authorities. Write a program that checks the income task declarations on consistency
- Represent the tax declarations with graphs.
  - How many graphs will you get?
  - How big are they?
  - How much memory do you need at least?

First Idea

- Write a context free grammar for the tax declarations
- From that, derive a XML schema
  - Enrich the grammar nonterminals with attributes
- Check context free structure of the tax declarations with the XML parser (contextfree consistency)
- This is usually assured by the tax form
  - It is, however, nevertheless necessary, if the forms have been fed into a computer, to avoid feeding problems.

Second Idea

- Write queries that checks document-local, but global constraints
  - Are there bills for all claimed tax reductions?
  - Are the appendices consistent with the main tax document?
- Global Constraints over all tax Declarations:
  - Have all bills for all claimed tax reductions really been payed by the tax payer?
  - Is a reduction for a debt reduced only once per couple?
  - ....
- Write an OCL *invariant specification* for the tax UML class diagram that checks the constraints
  - Use the Dresden OCL toolkit to solve the problem http://dresden-ocl.sf.net

Third Idea: Use Ontology Language

- OWL (description logic) can be used for consistency constraints, also of tax declarations
  - Result: a *tax ontology*, a vocabulary of classes with consistency constraints
  - OWL engines (RACER, Triple) can evaluate the consistency of tax specifications
  - Ontologies can formulate consistency criteria for an entire administrative workflow [Aßmann2005]
- Ontologies union a class specification (T-box) and an object base (A-box)
  - Classes are sets of objects
  - Classes need not have a unique name (no unique name assumption)
  - Objects can be members of several classes (no unique membership)
- Ontology services:
  - Subsumption checking (is a class subclass to another class)
  - Consistency checking (is an object member of several disjoint classes)
  - Satisfiability checking (is a class not a subclass of the empty class (empty set))

Example 3: UML Specifications in Software Engineering

- Imagine a UML model of the Java Development Kit JDK.
  - 7000 classes
  - Inheritance tree on classes
  - Inheritance lattice (dag) on interfaces
  - Definition-use graph: how big?
- Task: You are the release manager of the new JDK 1.8. It has 1000 classes more.
  - Ensure consistency please. - How?

Ideas

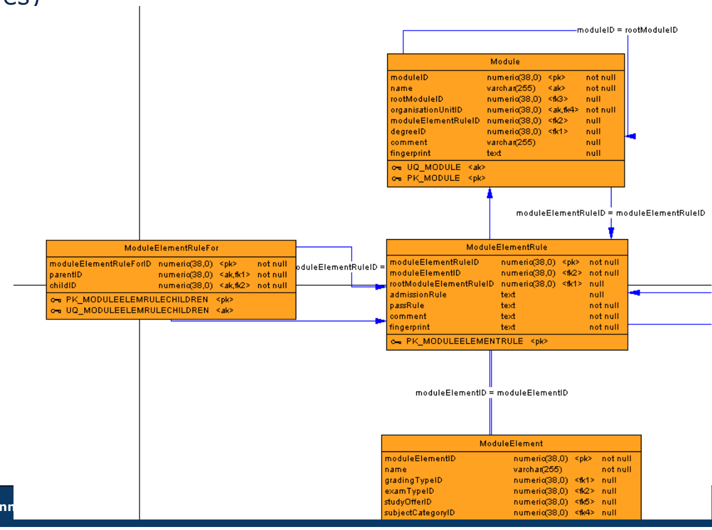
- Build up inheritance graphs and definition-use graphs
  - in a database
- Use F-Datalog for inheritance analysis
- Use OWL for inheritance analysis
- Analyse conditions such as
  - Depth of inheritance tree: how easy is it to use the library?
  - Hot-spot methods and classes: Most-used methods and classes (e.g., String)
    - Optimize them
  - Does every class/package have a tutorial?
  - Is every class containt in a roadmap for a certain user group? (i.e., does the documentation explain how to use a class?)

Example 3: Exam Enrollment

- Check if a student can enroll to a lecture
- Check if a student has passed his master degree

First Idea

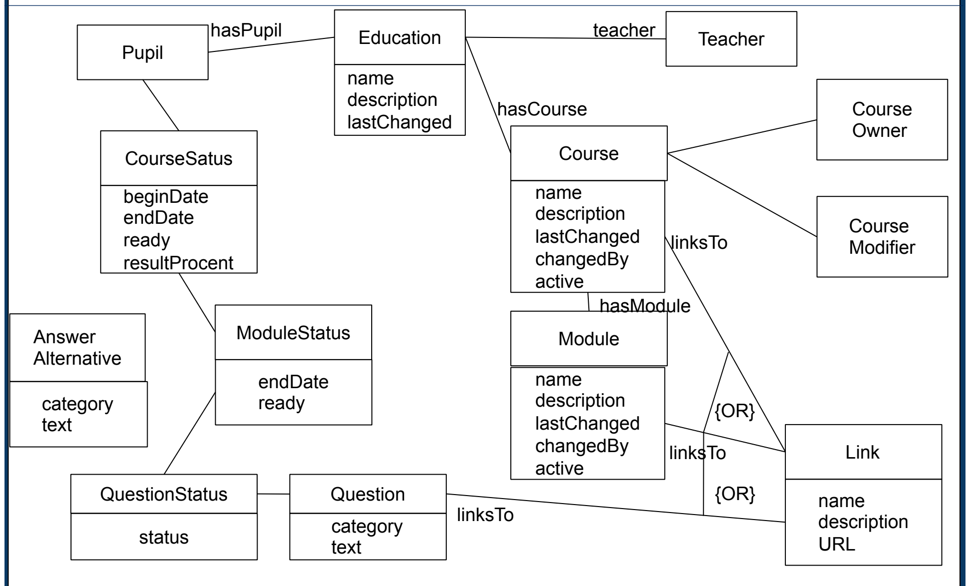
- Store all basic claims data in the database
- Write all contrains and rules into code fragments and check (stored procedures)



Second Idea

- Check all rules with Prolog or Datalog:
- attendMEMax(STUDENTID,MEID,N):-setof(A,nr (A,STUDENTID,MEID), L), length(L, N).
- attendAdditionalMax(STUDENTID,MEID,N):-setof(A,r (A,STUDENTID,MEID), L), length(L, N).
- attendModulesMax(STUDENTID,L,IMAX):-setof(MEID, (attendMEMax(STUDENTID,MEID,N),N>=IMAX,member (MEID,L)),LIST).
- attendModuleElementsMax(STUDENTID,L,IMAX,MAX):-setof(MEID, (attendMEMax(STUDENTID,MEID,N),N>=IMAX,member (MEID,L)),LIST),length(LIST, N),N>MAX.
- recommendGradingValues(STUDENTID,[K1|[]],N):-if_then_elseME (me(K1,B),K1,B),if_then_elseMEPASS(p(STUDENTID,K1),Y,B,0),N is Y.
- recommendGradingValues(STUDENTID,[K1|Rest],MIN):- recommendGradingValues(STUDENTID,Rest,X),if_then_elseME(me (K1,B),K1,B),if_then_elseMEPASS(p(STUDENTID,K1),Y,B,0),N is Y + X,N>=MIN,!.

Third Idea: use OWL on domain model
Example: The Domain Model of the Web-Based Course System



- Step 1: encode the diagram into a Datalog/DL fact base
- Step 2: specify integrity constraint rules
- Step 3: let the rules run

```
// Step 1: factbase
teacher(programming,john).
hasCourse(programming, lisp).
hasPupil(programming, mary).
hasModule(lisp,closures).
linksTo(linkA, closures).
linksTo(linkA, lisp).
linksTo(linkA, q).

// Step 2: integrity constraints specification
consistent(Link,Course,Module,Question) :-
   linksTo(Link,Course) ||
   linksTo(Link, Module) ||
   linksTo(Link,Question).
```

```
// Step 3: consistency checking query
:- consistent(linkA,lisp,closures,q)

// answer:
false
```


Third Idea: Use Ontology Language

- OWL (description logic) can be used for consistency constraints, also of UML domain models
  - Result: a *domain ontology*, a vocabulary of classes with consistency constraints about the domain
  - OWL engines (RACER, Triple) can evaluate the consistency of such domain specifications
  - Ontologies can formulate consistency criteria for domain models of applications and product lines [Aßmann2005]

The End: What Have We Learned

- Graphs and Logic are isomorphic to each other
- Using logic or graph rewrite systems, models can be validated
  - Analyzed
  - Queried
  - Checked for consistency
  - Structured
- Applications are many-fold, using all kinds of system relationships
  - Consistency of UML class models (domain, requirement, design models)
  - Structuring (layering) of USES relationships
- Logic and graph rewriting technology involves reachability questions

> Logic and edge addition rewrite systems are the Swiss army knifes of the validating modeler