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

1. Big Models
2. Examples of Graphs in Models
3. Types of Graphs
4. Analysis of Graphs in Models
   1. Layering of Graphs
   2. Searching in Graphs
   3. Checking UML Models with Datalog
   4. Transitive Closure
5. Further Examples
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, Macaszek 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
- 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](http://dx.doi.org/10.1007/11785477_2)
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.


http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/personen/juergen-ebert/juergen-ebert/


Graph rewriting for programs and models:

- U. Aßmann. Graph Rewrite Systems for Program Optimization. ACM Transactions on Programming Languages and Systems, June 2000.
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]

5.1 THE PROBLEM: HOW TO MASTER LARGE MODELS
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

5.2 GENERATING GRAPHS FROM MODELS AND SOFTWARE
All Specifications Have an Internal Graph-Based Representation

- Texts are parsed to abstract syntax trees (AST)
  - Two step procedure
    - Concrete Syntax Tree
    - Abstract Syntax Tree
- 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)
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 )
```
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 )

```
Expr
   /
  /|
 ( | )
Expr & & Expr
   /
  /|
 ( | )
Expr || Expr
   /
  /|
 ( | )
Expr ! Expr
   /
  /|
 ( | )
Expr
   /
  /|
 ( | )
Expr
   /
  /|
 ( | )
Expr

Var id = found

Var id = looking

true
```
From the CST to the AST

TU Dresden, Prof. U. Ahmann
- 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)
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
Def-Use Graphs (DUG) and Use-Definition-Use Graphs (UDUG)

- Every language and notation has
  - Definitions of items (definition of the variable Foo)
  - 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

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

```
VarDecl
  type=boolean
  id=looking

VarDecl
  Type=boolean
  id=found

IfStmt
  &&
  Block

VarName
  id=looking

VarName
  id=found

Block
looking
!
found
```
UDUGs are used in refactoring operations (e.g. renaming a class).

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)
Refactor the name Person to Human:

class Person { .. }                          Definition

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

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

```
while
  if
    a+=5;
    print a
    return
  while
    print a++
```
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

```
Flow-Insensitive

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

TU Dresden, Prof. U. Aßmann
Model Consistency
```
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
All diagram sublanguages of UML generate internal graph representations
- They can be analyzed and checked with graph techniques

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)
- 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
5.3 TYPES OF GRAPHS IN SPECIFICATIONS

Lists, Trees, Dags, Graphs
Structural constrains on graphs
(background information)
We deal here mostly with directed graphs (digraphs)
- 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 algorithm
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 $k$
- One source (root)
- One sink
- Every other node has fan-in 1, fan-out 1
- Represents a *total order* (sequentialization)
- Gives
  - Prioritization
  - Execution order
- One source (root)
- Many sinks (leaves)
- Every node has fan-in $\leq 1$

- **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)

- Graph with cycles, however, only between sisters
  - No cycles between hierarchy levels
- Graph can be “reduced” to one node
- Advantage
  - Tree can be used as a conceptual hierarchy
- Example
  - UML statecharts
  - Control-flow graphs of Modula, Ada, Java (not C, C++)
  - SA data flow diagrams
Reduction of a Reducible Graph

B1 → B2 → B3 → B4

B1a → B3a

B1a → B3a → B1b
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 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

List: strong assertion: total order  Sequential

Tree: still abstraction possible  Hierarchies

Dag: still layering possible  Partial order

Layered

Graph: the worst case  Unstructured

Ease of Understanding
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
Structuring Improves Worst Case

- List: strong assertion: total order
  - Sequential
- Tree: still abstraction possible
  - Hierarchies
- Dag: still layering possible
  - Partial order
  - Layered

Ease of Understanding

- Structured graph
- Graph with analyzed features
  - Unstructured
- Graph: the worst case
  - Unstructured
5.4 METHODS AND TOOLS FOR ANALYSIS OF GRAPH-BASED MODELS
The Graph-Logic Isomorphism

- 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

Carl Gustav

married

father

Victoria

married

mother

Silvia

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

- Graphs can be noted textually
- Graphs consist of nodes, relations
- Relations link nodes
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
5.4.1 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
    - Layered USES relationship

- A *layered architecture* is the dominating style for large systems
- Outer, upper layers use inner, lower layers (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?

- GustavAdolf
- Carl Gustav
- Sibylla
- Madeleine
- Silvia
- Adam
- Walter
- Alice
- Desiree
- Victoria
- Ralf
Pattern and Rules

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

\[
isParentOf(\text{Parent}, \text{Child1}) \land isParentOf(\text{Parent}, \text{Child2})
\]

- And define the rule
  
  \[
  \text{if } isParentOf(\text{Parent}, \text{Child1}) \land isParentOf(\text{Parent}, \text{Child2}) \text{ then } sameGeneration(\text{Child1}, \text{Child2})
  \]
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
5.4.2 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
  - reachabilities 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)
```
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).
# 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
Description Logic (DL)

- A special form of *typed binary* Datalog (typed EARS)
  - Only with unary (classes) and binary relations (relationships)
  - Classes and objects, instead of untyped Datalog constants
  - Relationship types and relations
  - Inheritance of classes and relationships
  - All knowledge is specified with *triples*, simple sentences of Verb-Predicate-Object
  - OWL (Web Ontology Language)

Adam `instanceOf` Person.
Sibylla `instanceOf` Person.
GustavAdolf `instanceOf` Person.
King `isA` Person.
Princess `isA` Person.
...
Adam `parentOf` GustavAdolf.
Adam `parentOf` Sibylla.
...
Datalog, DL, OCL, and EARS:
Extended Relational Algebra

- 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
  - Datalog is a simple variant of Prolog
- DL languages:
  - OWL (ontology web language)
  - OIL (ontology interchange language)
- OCL does not have transitive closure, but iteration
Datalog, DL, OCL, and EARS: Extended Relational Algebra

- Relational Algebra (SQL)
- Description Logic (OWL)
- Binary Datalog (EARS)
- Datalog (with recursion) (SQL3)
- Datalog with negation and recursion
- Prolog with negation and recursion
- "Business rules" (decidable)

OCL

classes = unary predicates

binary predicates
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
    - Data-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
5.4.3 EXAMPLE FOR MODEL VALIDATION: CHECKING UML DIAGRAMS WITH DATALOG

- Step 1: encode the diagram into a Datalog or DL fact base
- Step 2: define integrity constraint rules
- Step 3: let the rules run
Example: The Domain Model of the Web-Based Course System

- **Pupil**
  - hasPupil

- **Course Status**
  - beginDate
  - endDate
  - ready
  - resultPercent

- **Answer**
  - Alternative
    - category
    - text

- **Question Status**
  - status

- **Question**
  - category
  - text

- **Module Status**
  - endDate
  - ready

- **Module**
  - hasModule
  - {OR}

- **Link**
  - name
  - description
  - URL

- **Course**
  - name
  - description
  - lastChanged
  - changedBy
  - active

- **Teacher**
  - teacher
  - hasCourse

- **Course Owner**

- **Course Modifier**

- **TU Dresden, Prof. U. Aßmann**
// Step 1: construct fact base: the UML class diagram
// in Datalog fact syntax.
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
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
  - Interpret tree and links as relations
  - Graph algorithms and queries can be applied to the web!
RDF (resource description framework, a simple graph language)
OWL (description logic, www.w3c.org) adds classes, inheritance, inheritance on binary relations, expressions and queries on binary relations
Other experimental languages SPARQL (Manchester), Flora/XSB (NY Stony Brook, www.ontoprise.com), Florijd (Freiburg)
New languages are being developed
  - In the European network REWERSE (www.rewerse.net)
The Swiss-Knife of Graph Analysis

5.4.4 REACHABILITY QUERIES WITH TRANSITIVE CLOSURE
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**
  \[ \text{descendant}(V,N) :\text{-} \text{isChildOf}(V,N). \]

- **Transitive rule (recursion rule)**
  - **left recursive:**
    \[ \text{descendant}(V,N) :\text{-} \text{descendant}(V,X),\text{isChildOf}(X,N). \]
  - **right recursive:**
    \[ \text{descendant}(V,N) :\text{-} \text{isChildOf}(V,X),\text{descendant}(X,N). \]
Impact of Recursion Rule

Impact only shown for Adam, but is applied to other nodes too.
PP Path Problems:
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

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

- **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),superScope(C,B).`
  - `unreachable(A,B)`

- **Answer the queries**
  - `? superClass(Man,X)`
  
  `>> {X=Person,X=Human}`

  - `? superClass(Woman,Y)`
  
  `>> {Y=Person,Y=Human}`
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 works on one relation
- 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
Any Datalog program or EARS graph rewrite system can be transformed into an equivalent one
- That is free of 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)
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
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 $O(n^3)$ (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.
Transitive closure is a general graph operator
- Computing reachability
- Can be applied generically to all relations!

Many other Datalog rule systems are also generic
- 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“

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>System.out.println</td>
</tr>
<tr>
<td>main</td>
<td>compute</td>
</tr>
<tr>
<td>main</td>
<td>twice</td>
</tr>
<tr>
<td>main</td>
<td>add</td>
</tr>
<tr>
<td>compute</td>
<td>twice</td>
</tr>
<tr>
<td>compute</td>
<td>add</td>
</tr>
</tbody>
</table>
When a specification becomes big...

5.4.5 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 [Abmann2005]
- 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](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 \([\text{Aßmann2005}]\)
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 contained 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
- Store all basic claims data in the database
- Write all constrains 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
Consistency Checking on UML Class Diagrams with Description Logic

- 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 graph rewrite systems are the Swiss army knife of the validating modeler