TECHNISCHE UNIVERSITÄT DRESDEN

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

12) Validation of Graph-Based Models and Programs (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
- WS14/15, 24.11.2014

- **Lecturer**: Dr. Sebastian Götz

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
5. Transitive Closure and Reachability

Contents

- Different kinds of relations: Lists, Trees, DAGs, Graphs
- The **graph-logic isomorphism**
- Analysis, querying, searching graph-based models
    - The “Same Generation” Problem
    - Datalog and Edge Addition Rewrite Systems (EARS)
    - Transitive Closure
- Consistency checking of graph-based specifications (aka model validation)
    - Projections of graphs
    - Transformation of graphs

- Understand that software models can become very large
  - the need for appropriate techniques to handle large models
  - the need for automatic analysis of the models

- Learn how to use graph-based techniques to analyze and check models for consistency, well-formedness and integrity
  - Datalog,
  - Graph Query Languages,
  - Description Logic,
  - Edge Addition Rewrite Systems and
  - Graph Transformations.

- Understand some basic concepts of simplicity in software models

Motivation

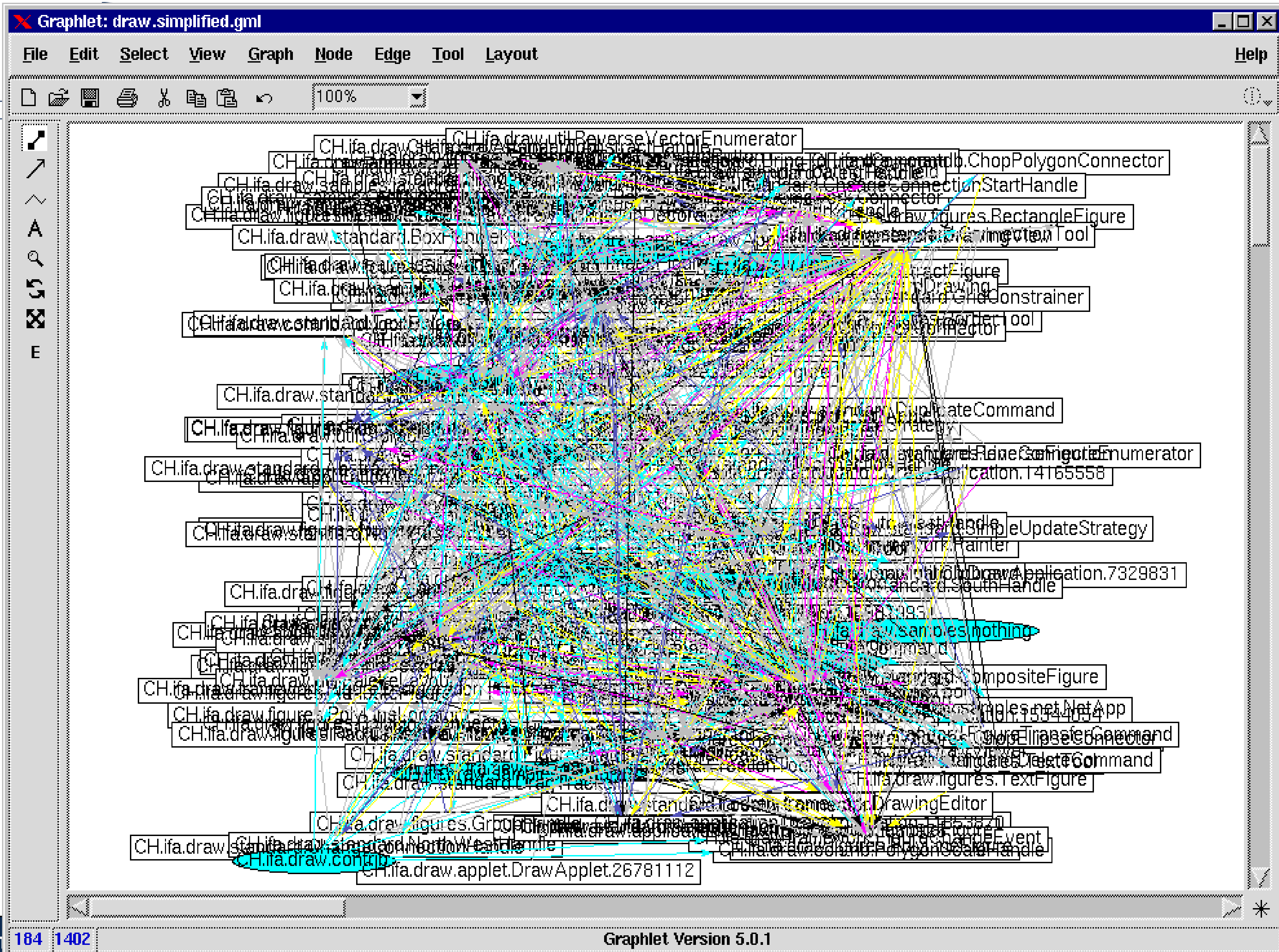
- Software engineers must be able to
  - handle ***big*** design specifications (design models) during development
  - work with ***consistent*** models
  - ***measure*** models and implementations
  - ***validate*** models and implementations

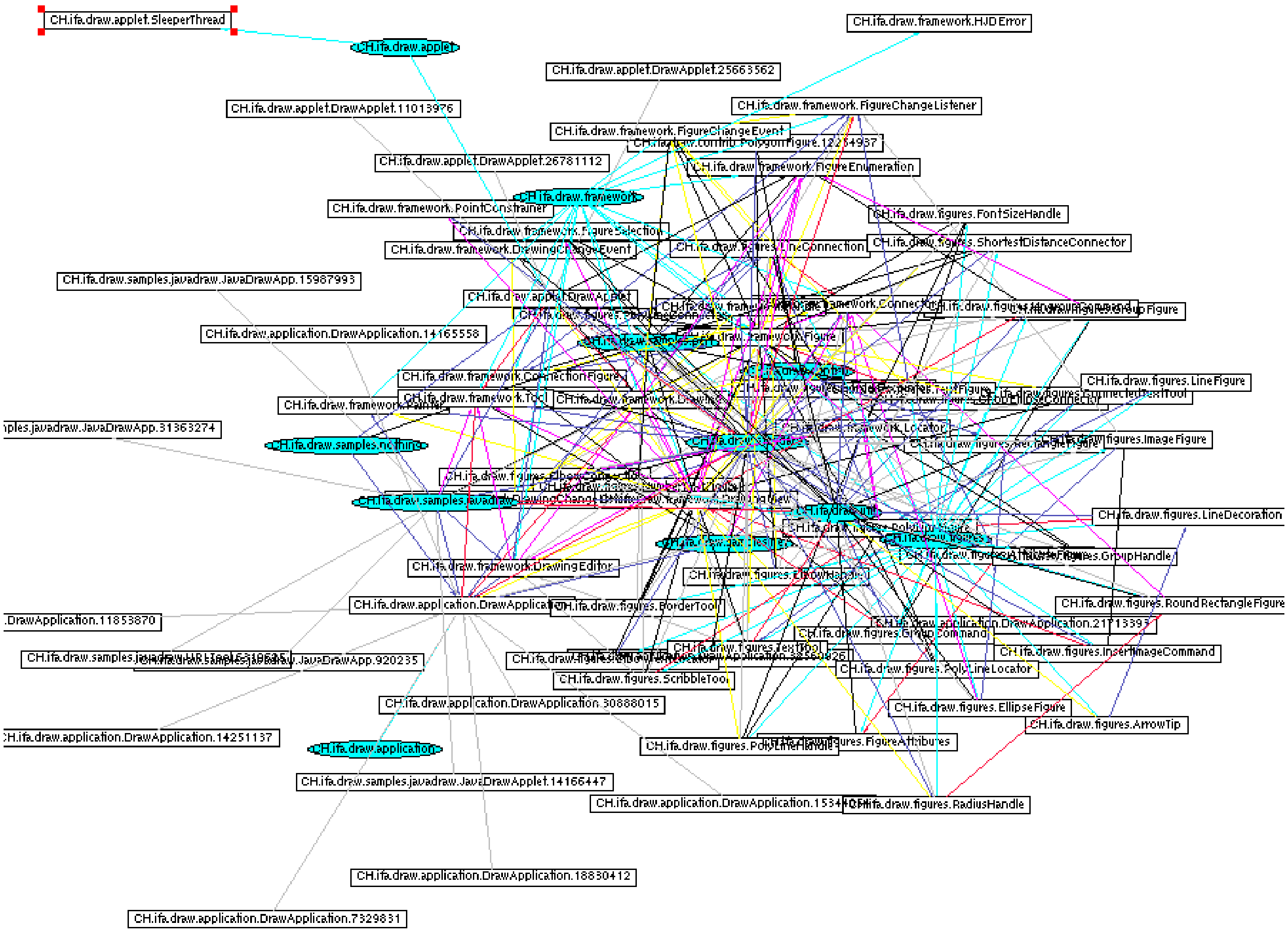
- Real models and systems become very complex
- Most specifications are graph-based
  - We have to deal with basic graph theory to be able to measure well

- Large models have large graphs
- They can be hard to understand

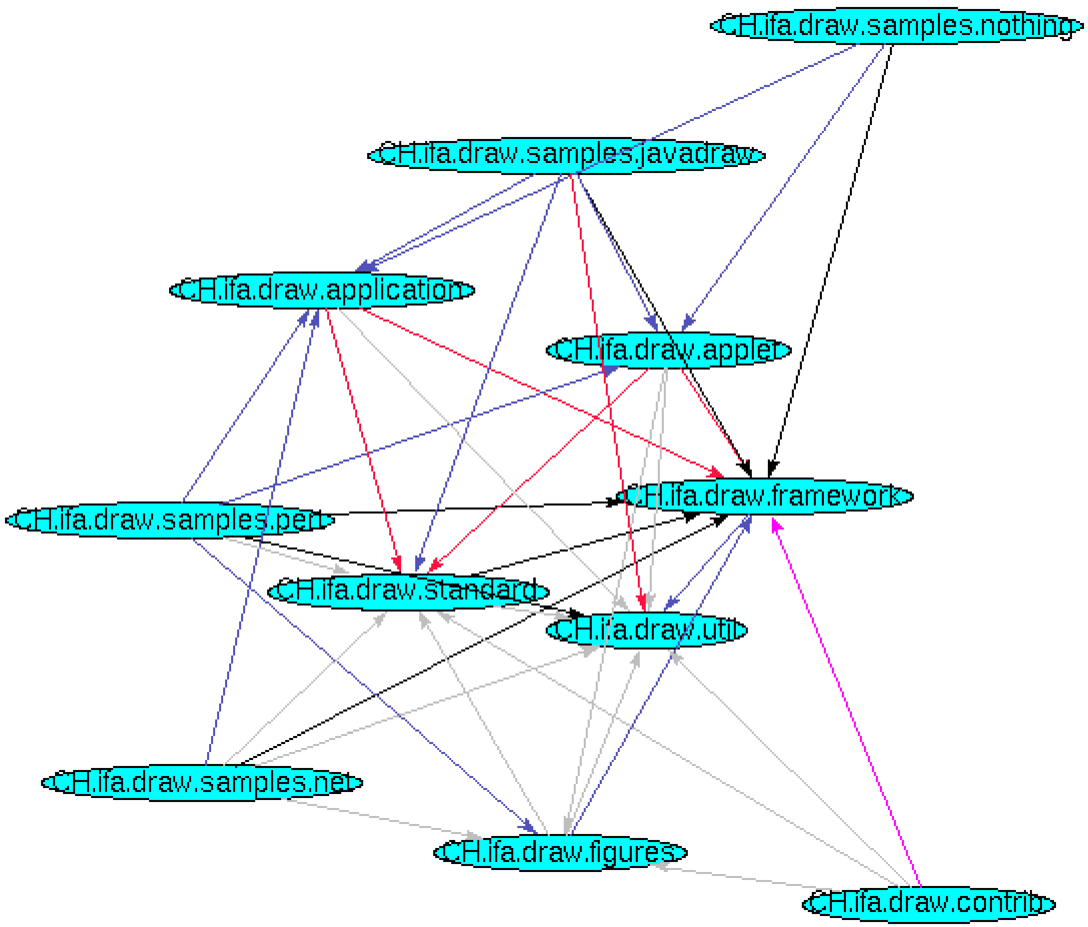
Figures taken from Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe]

12.1 THE PROBLEM: HOW TO MASTER LARGE MODELS


Graphlet: draw.simplified.gml
File
Edit
Select
View
Graph
Node
Edge
Tool
Layout
Help
100%
CH.ifa.draw.util.ReverseVectorEnumerator
CH.ifa.draw.standard.BoxHandleKit
CH.ifa.draw.standard.DuplicateCommand
CH.ifa.draw.standard.CompositeFigure
CH.ifa.draw.figures.TextFigure
CH.ifa.draw.figures.RectangleFigure
CH.ifa.draw.standard.GridConstrainer
CH.ifa.draw.applet.DrawApplet.26781112
CH.ifa.draw.contrib
CH.ifa.draw.samples.nothing
184
1402
Graphlet Version 5.0.1


CH.ifa.draw.applet.SleeperThread
CH.ifa.draw.applet
CH.ifa.draw.framework.HJDError
CH.ifa.draw.applet.DrawApplet.25663562
CH.ifa.draw.applet.DrawApplet.11013976
CH.ifa.draw.framework.FigureChangeListener
CH.ifa.draw.framework.FigureChangeEvent
CH.ifa.draw.applet.DrawApplet.26781112
CH.ifa.draw.framework.FigureEnumeration
CH.ifa.draw.framework
CH.ifa.draw.framework.PointConstrainer
CH.ifa.draw.figures.FontSizeHandle
CH.ifa.draw.framework.FigureSelection
CH.ifa.draw.framework.DrawingChangeEvent
CH.ifa.draw.figures.LineConnection
CH.ifa.draw.figures.ShortestDistanceConnector
CH.ifa.draw.samples.javadraw.JavaDrawApp.15987995
CH.ifa.draw.applet.DrawApplet
CH.ifa.draw.application.DrawApplication.14165558
CH.ifa.draw.framework.Figure
CH.ifa.draw.figures.LineFigure
CH.ifa.draw.framework.ConnectionFigure
CH.ifa.draw.framework.Tool
ples.javadraw.JavaDrawApp.31363274
CH.ifa.draw.framework.Locator
CH.ifa.draw.samples.nothing
CH.ifa.draw.figures.ImageFigure
CH.ifa.draw.samples.javadraw
CH.ifa.draw.figures.ElbowConnection
CH.ifa.draw.figures.LineDecoration
CH.ifa.draw.framework.DrawingEditor
CH.ifa.draw.figures.GroupHandle
CH.ifa.draw.figures.ElbowHandle
CH.ifa.draw.figures.RoundRectangleFigure
.DrawApplication.11853870
CH.ifa.draw.figures.BorderTool
CH.ifa.draw.figures.GroupCommand
CH.ifa.draw.samples.javadraw.JavaDrawApp.920235
CH.ifa.draw.figures.TextTool
CH.ifa.draw.figures.InsertImageCommand
CH.ifa.draw.figures.PolyLineLocator
CH.ifa.draw.figures.ScribbleTool
CH.ifa.draw.application.DrawApplication.30888015
CH.ifa.draw.figures.EllipseFigure
CH.ifa.draw.figures.ArrowTip
CH.ifa.draw.application
CH.ifa.draw.figures.FigureAttributes
CH.ifa.draw.samples.javadraw.JavaDrawApplet.14166447
CH.ifa.draw.figures.RadiusHandle
CH.ifa.draw.application.DrawApplication.18830412
CH.ifa.draw.application.DrawApplication.7329831

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

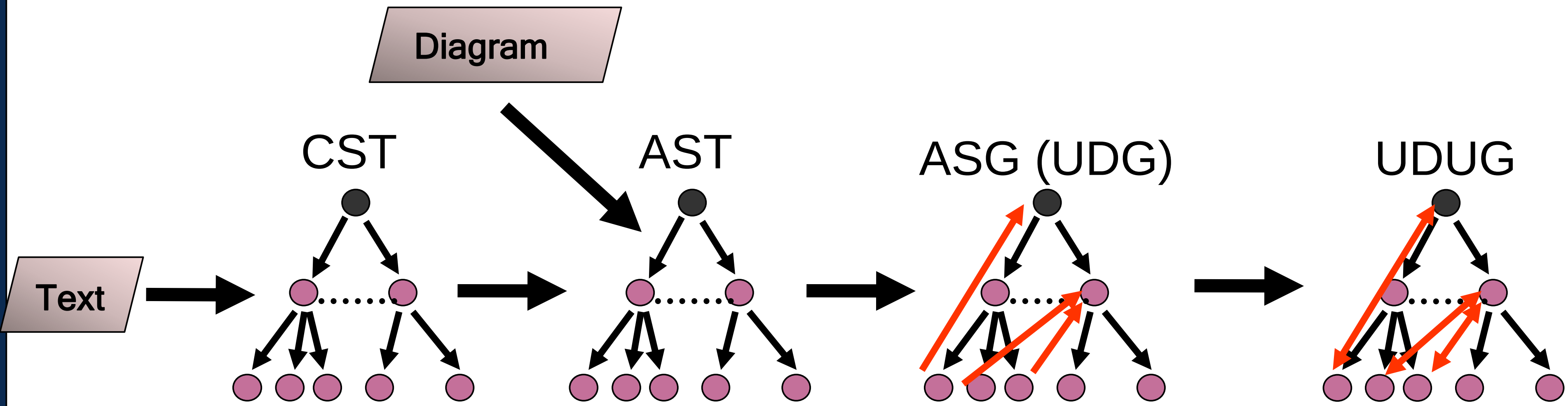
How are models and programs represented in a Software Tool?

Some Relationships (Graphs) in Software Systems

12.2 GENERATING GRAPHS FROM DIAGRAMS AND PROGRAMS

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)



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

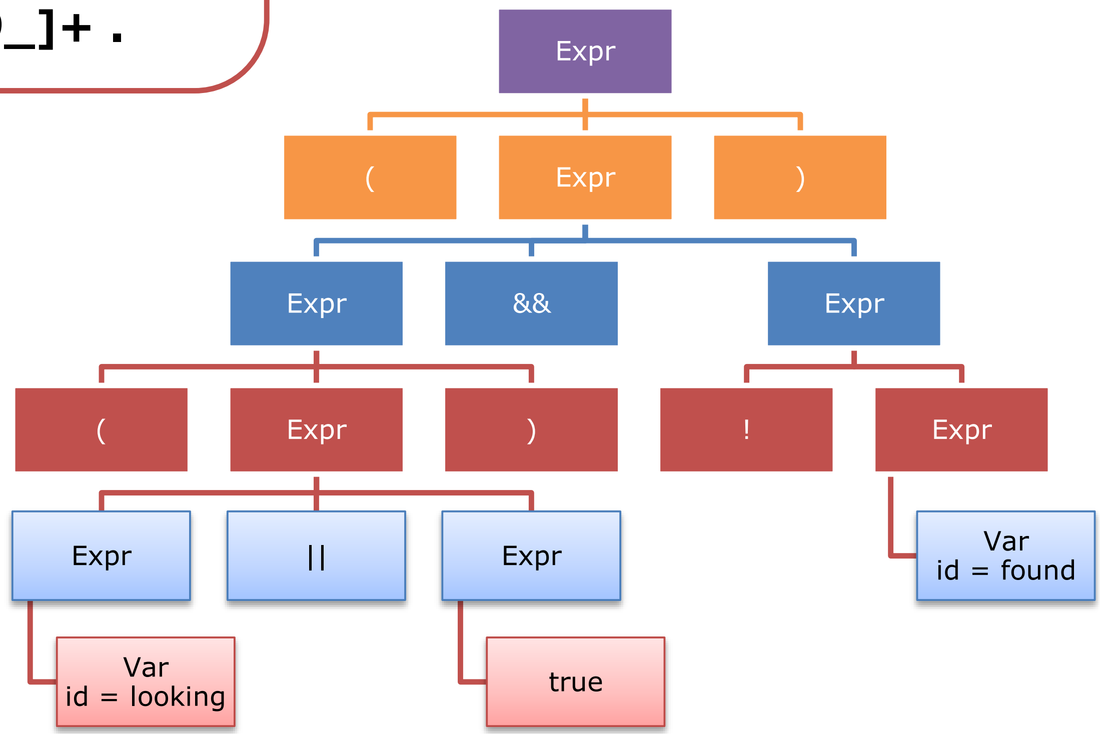
Parsing this string:

((looking || true) && !found)

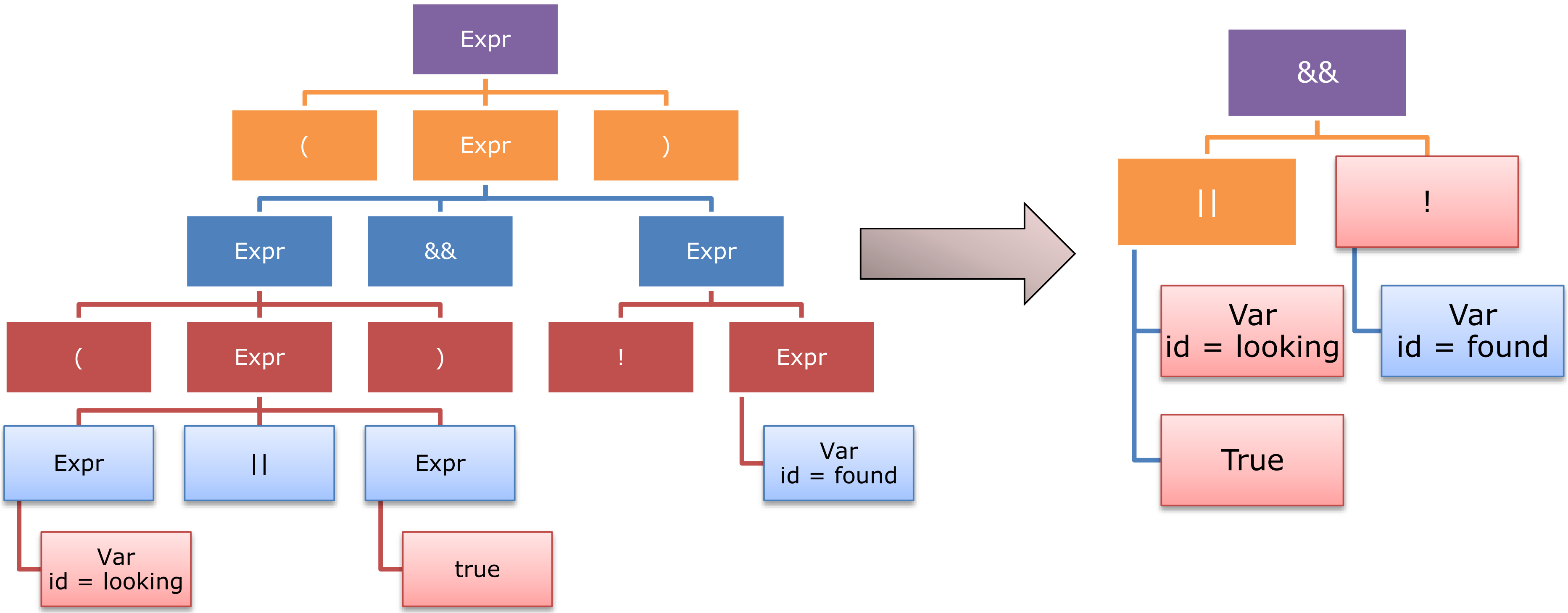
CST - Example

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

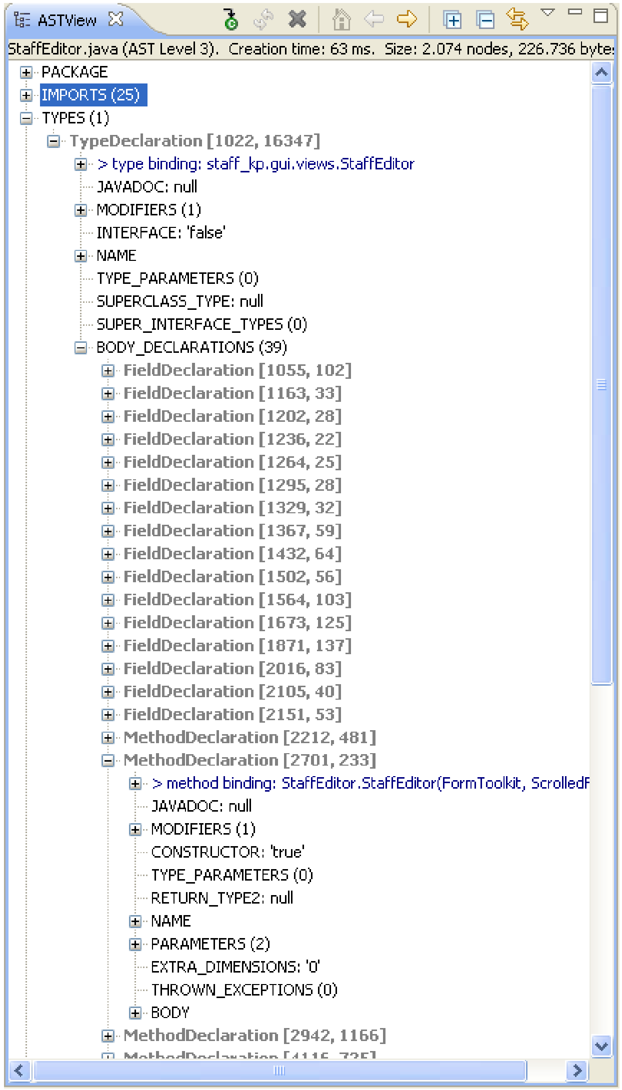
Parsing this string:
((looking || true) && !found)



From the CST to the AST



- 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)



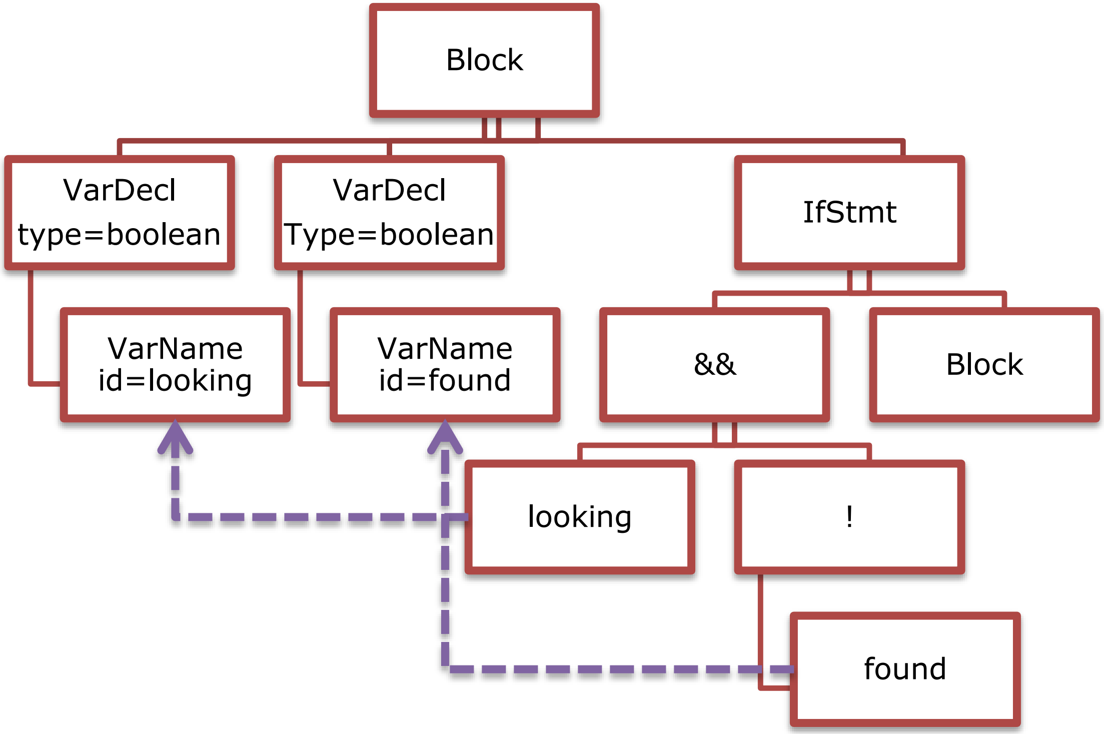
- Problem with ASTs: They do not support static semantic checks, re-factoring and browsing operations, e.g.:
  - Name semantics:
    - Have all used variables been declared? Are they declared once?
    - Have all classes used been imported?
  - Are the types used in expressions / assignments compatible? (type checking)
  - Referencing:
    - Navigate to the declaration of method call / variable reference / type
  - 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)

- Many languages and notations have
  - ***Definitions*** of items (definition of the variable `Foo`), which specify the 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 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
  - Type inference
  - Coercion (transfer to new instance of other type with same contents)

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");
}
```

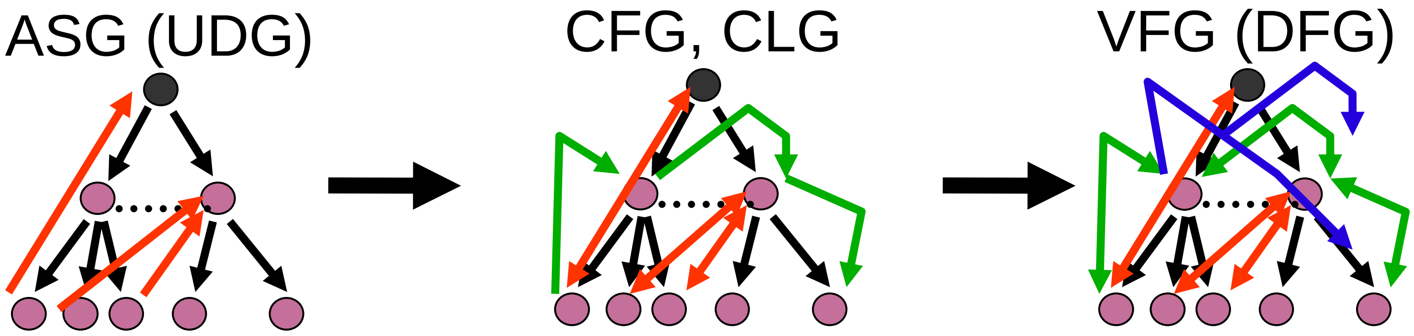
- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions
- Refactoring can be supported by tools
  - The Use-Def-Use-graph forms the basis of refactoring tools
- However, building the Use-Def-Use-Graph 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

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

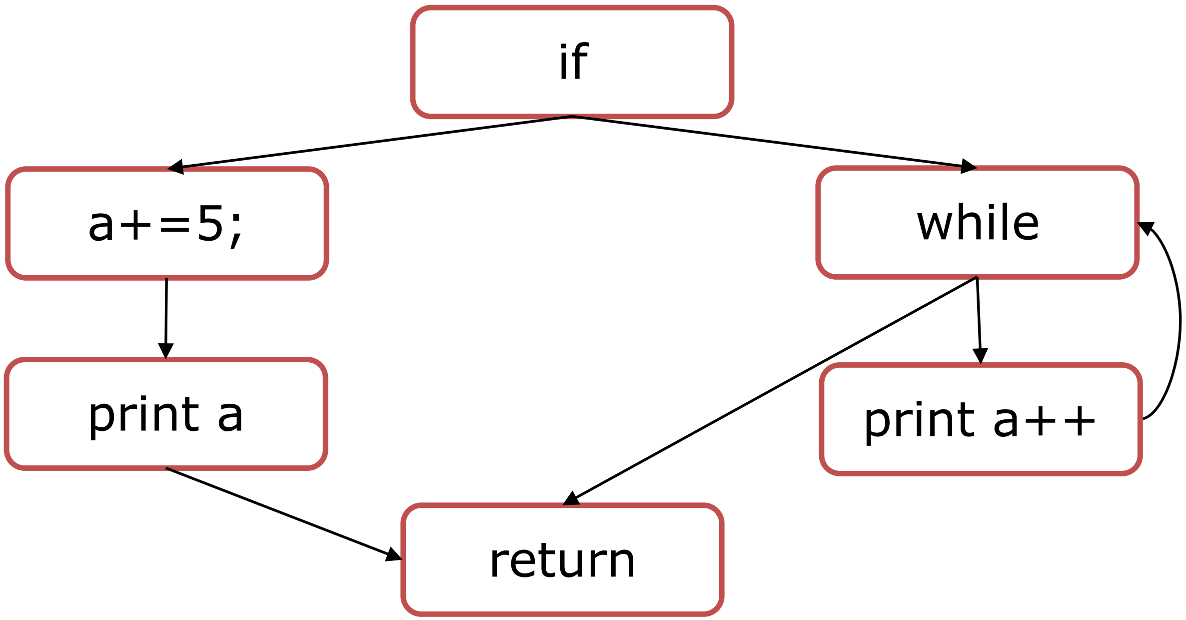
The same remarks holds for graphic specifications

- Hence, all specifications are graph-based!



Control-Flow Graphs

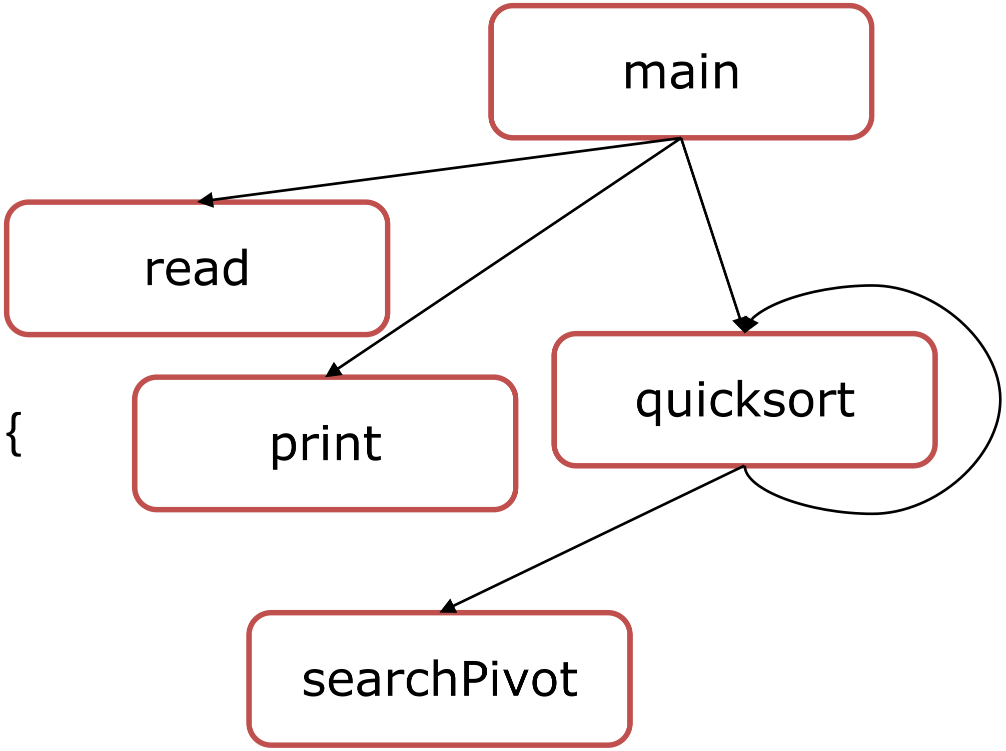
- Describe the control flow in a program
- Typically, if statements and switch statements split control flow
  - Their ends join control flow
- Control-Flow Graphs *resolve* symbolic labels
  - Perform name analysis on labels
- Nested loops are described by nested control flow graphs



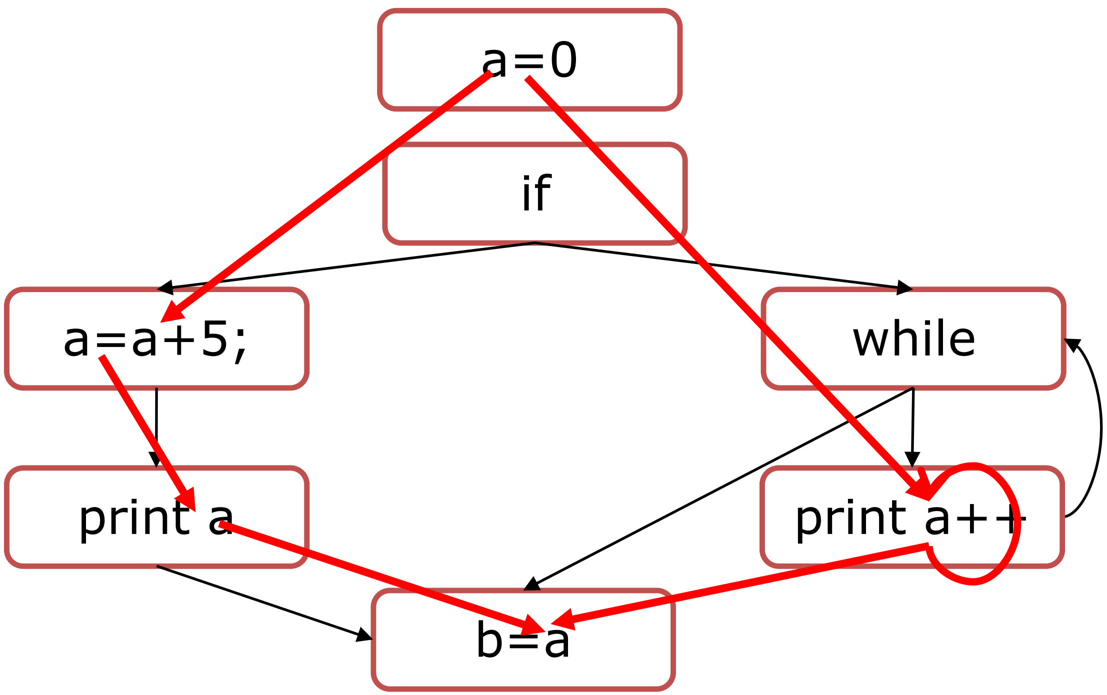
Simple (Flow-Insensitive) Call Graph (CLG)

- 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]);
}
```

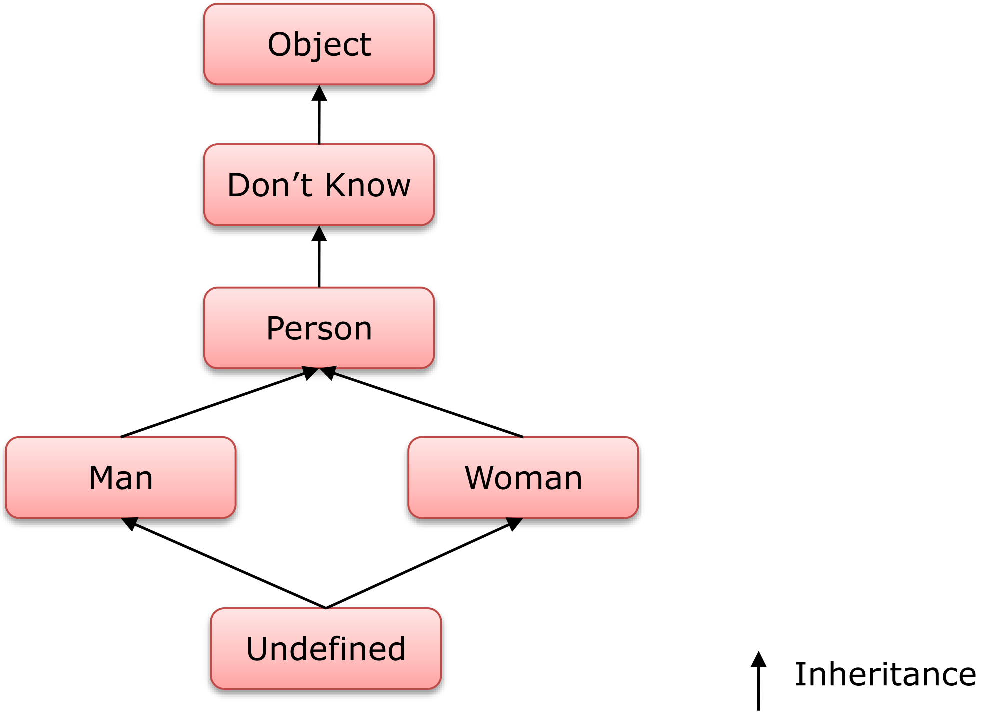


- A *data-flow graph (DFG)* aka *value-flow graph (VFG)* describes the flow of data through the variables
  - DFG are based on control-flow graphs
- Building the data-flow graph is called *data-flow analysis*
  - Data-flow analysis is often done by *abstract interpretation,* the symbolic execution of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



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



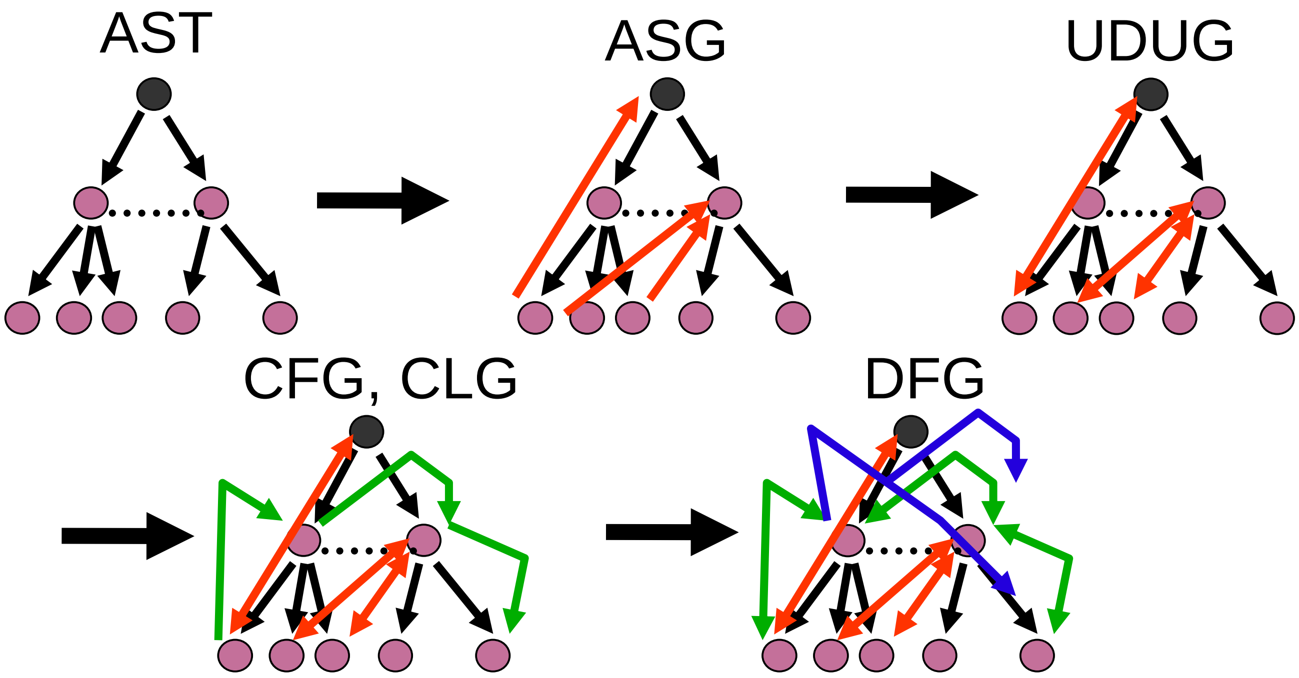
UML Graphs

- 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

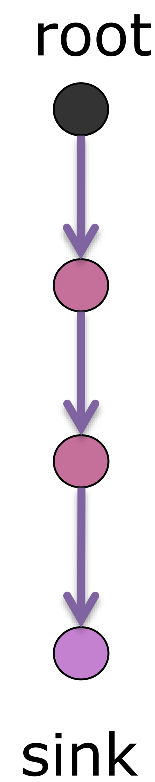
- In modeling, we deal mostly with *directed graphs (digraphs)* representing unidirectional relations
  - lists, trees, DAGs, overlay graphs, reducible (di-)graphs, graphs
- There are two different abstraction levels; we are interested in the logical level:
  - **Logical level** (conceptual, abstract, often declarative, problem oriented)
    - Methods to specify 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, binary decision diagrams (BDDs)
    - Imperative algorithms
    - Pointer based representations and algorithms

- Fan-in
  - In-degree of a 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 $p = (n_1, n_2, \ldots, n_k)$ 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)

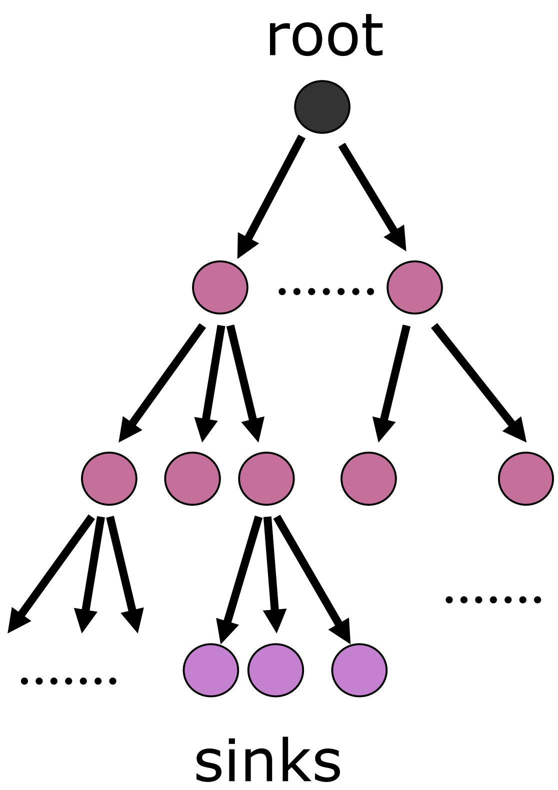
- Gives
  - Prioritization
  - Execution order



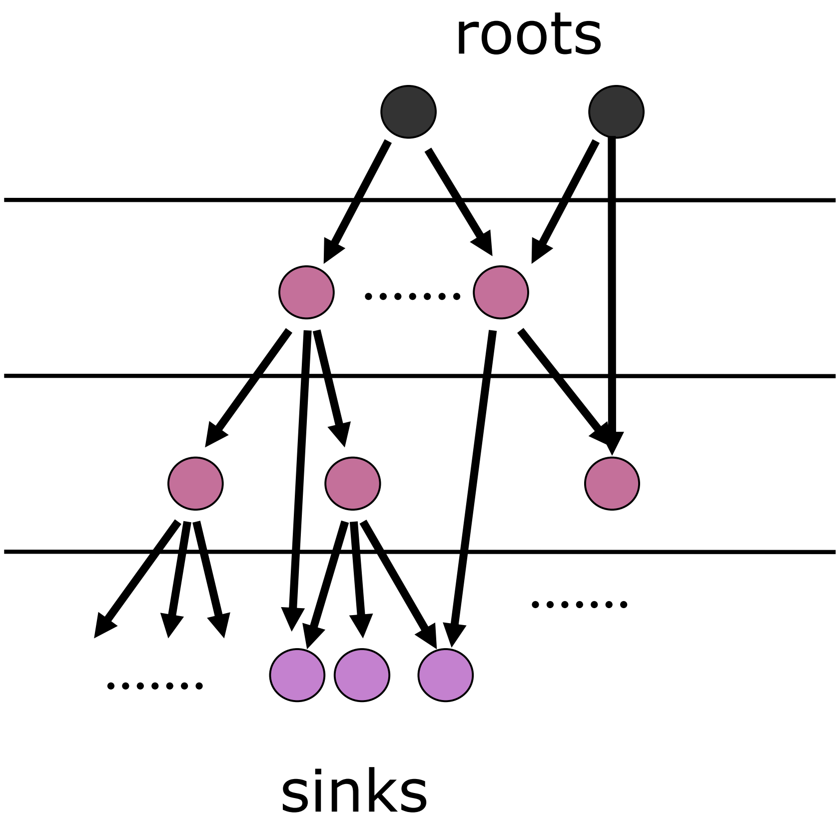
- One source (root)
- Many sinks (leaves)
- Every node has fan-in <= 1

- *Hierarchical abstraction*:
  - A node *represents* or *abstracts* all nodes of a sub tree

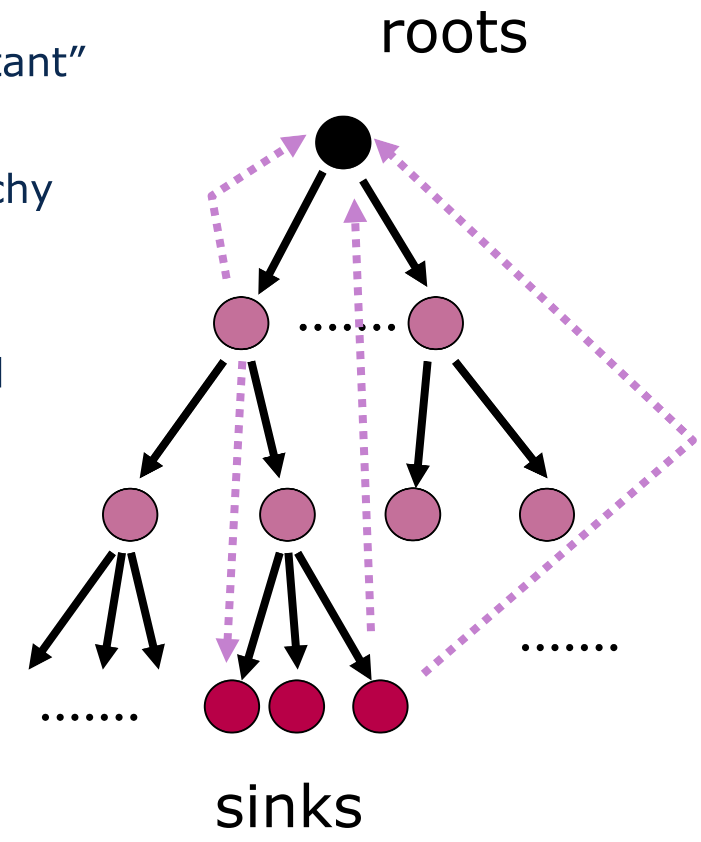
- Example
  - Structured Analysis (SA) function trees
  - Organization trees (line organization)



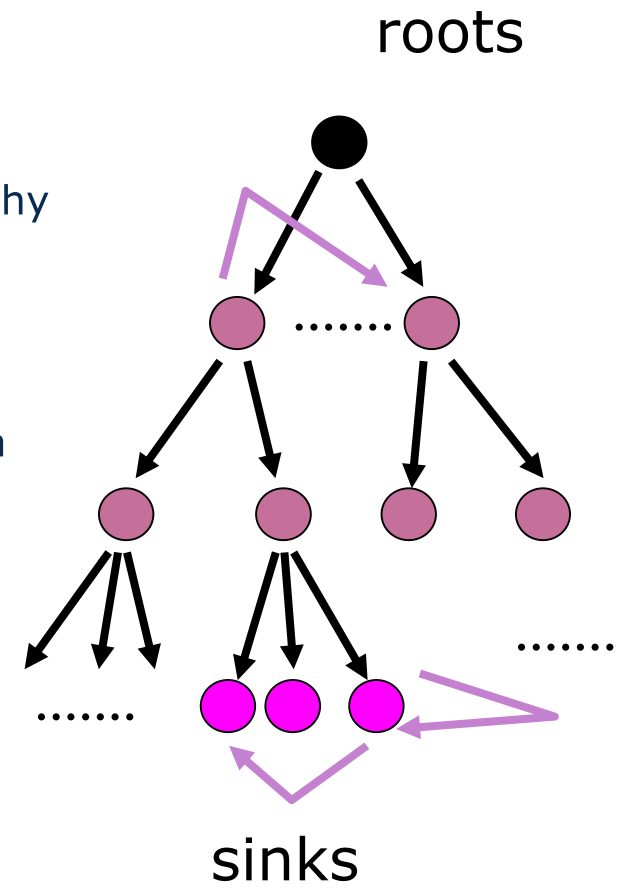
- Many sources
  - A jungle (term graph) is a dag with one root
- Many sinks
- Fan-in, fan-out arbitrary
- Represents a partial order
  - Less constraints than in a total order
- Weaker hierarchical abstraction feature
  - Can be layered
- Example
  - UML inheritance DAGs
  - Inheritance lattices



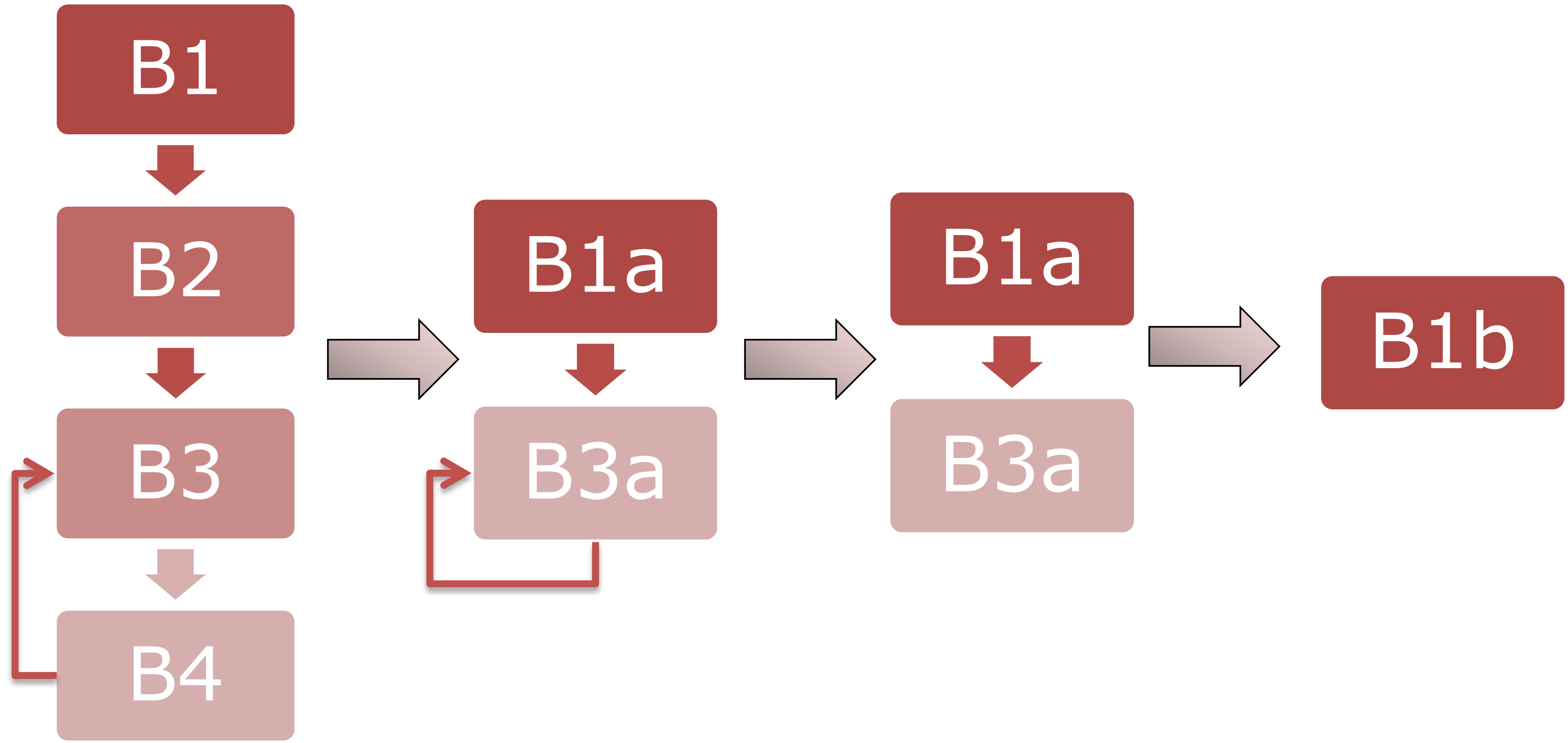
- Skeleton tree with **overlay graph** (secondary links)
  - Skeleton tree is primary
  - Overlay graph is secondary: “less important”
- Advantage of an Overlay Graph
  - Tree can be used as a conceptual hierarchy
  - References to other parts are possible
- Example
  - XML, e.g., XHTML. Structure is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



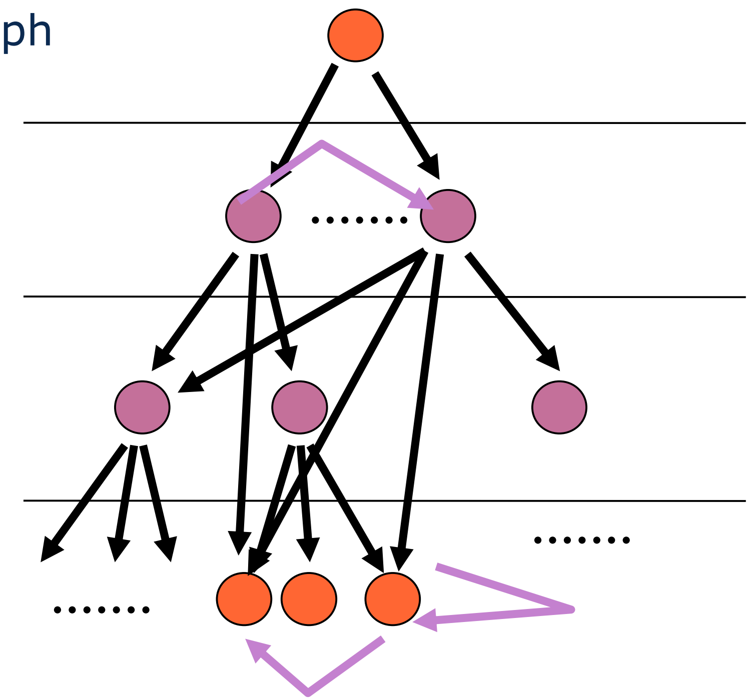
- A **reducible graph** is a graph with cycles, however, only between siblings
  - No cycles between hierarchy levels
- Graph can be “reduced” to one node
- Advantage
  - Tree can be used as a conceptual hierarchy
- Example
  - UML statecharts
  - UML and SysML component diagrams
  - Control-flow graphs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



Reduction of a Reducible Graph

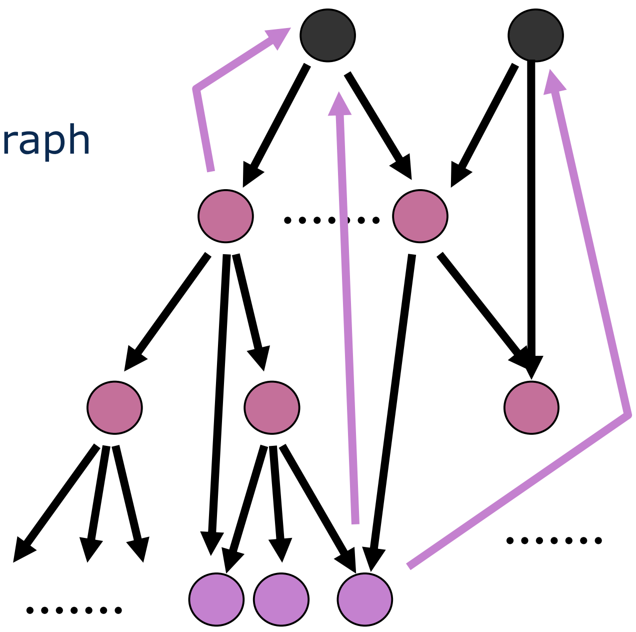


- Like reducible graphs, however, sharing between different parts of the skeleton trees
  - Graph cannot be “reduced” to one node
- Advantage
  - Skeleton can be used to layer the graph
  - Cycles only within one layer
- Example
  - Layered system architectures



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

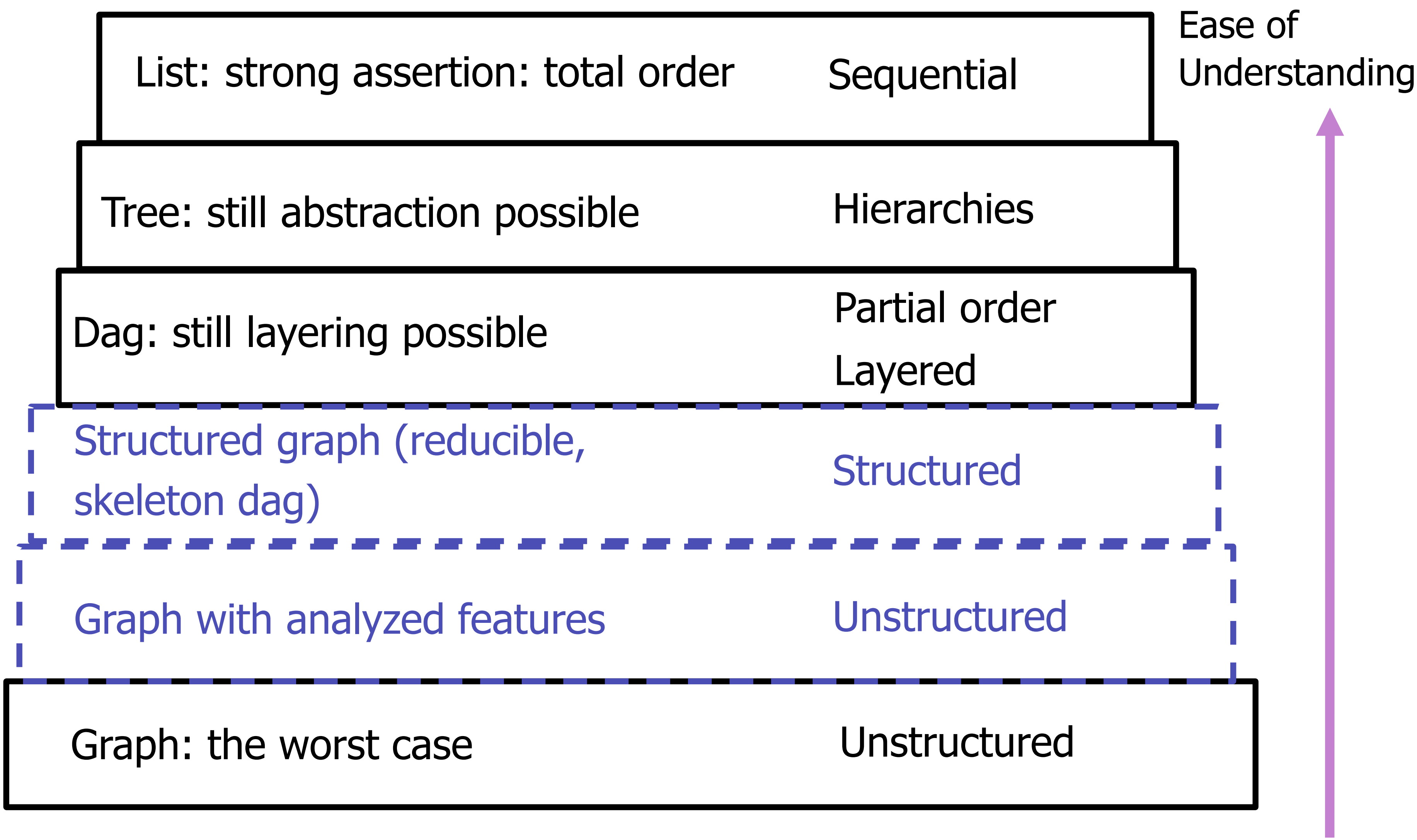
Ease of Understanding ↑

List: strong assertion: total order	Sequential
Tree: still abstraction possible	Hierarchies
Dag: still layering possible	Partial order Layered
Graph: the worst case	Unstructured

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

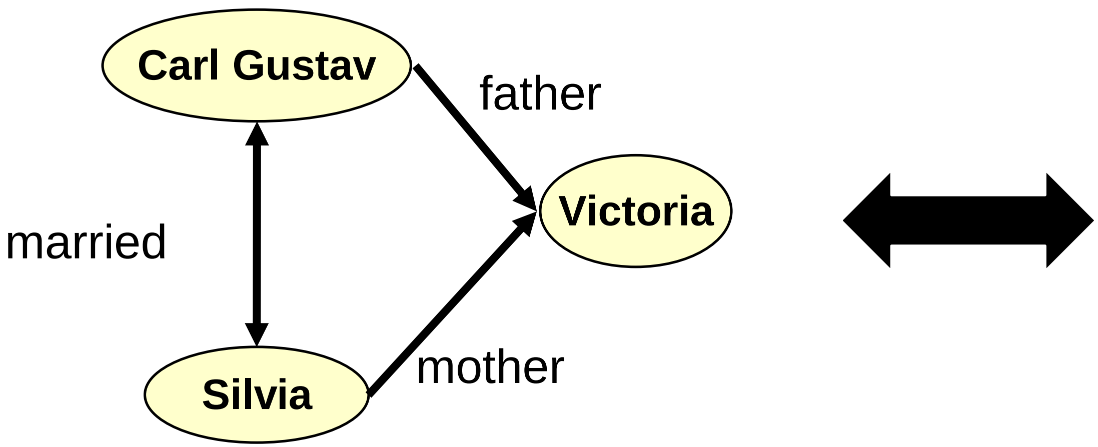
Structuring Improves Worst Case



12.4 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 over 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



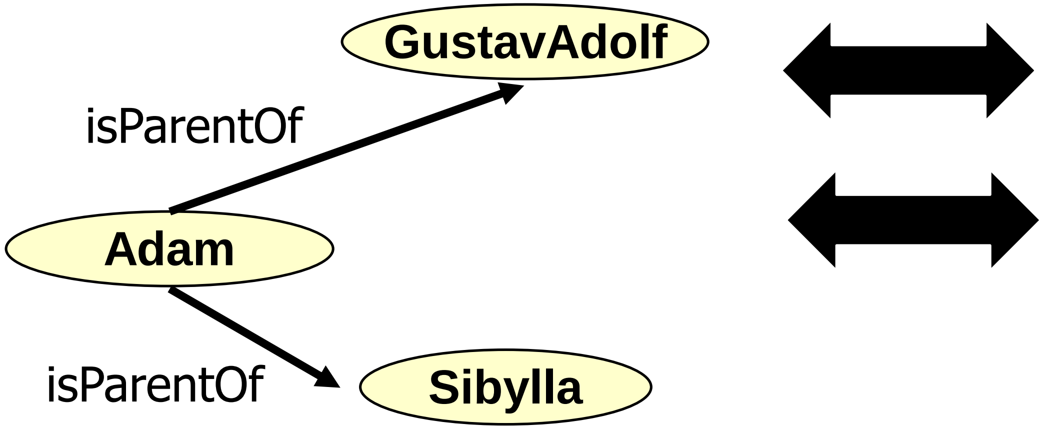
// fact base
married(CarlGustav,Silvia).
married(Silvia, CarlGustav).
father(CarlGustav,Victoria).
mother(Silvia,Victoria).

// Normalized English
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*):




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

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

- 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

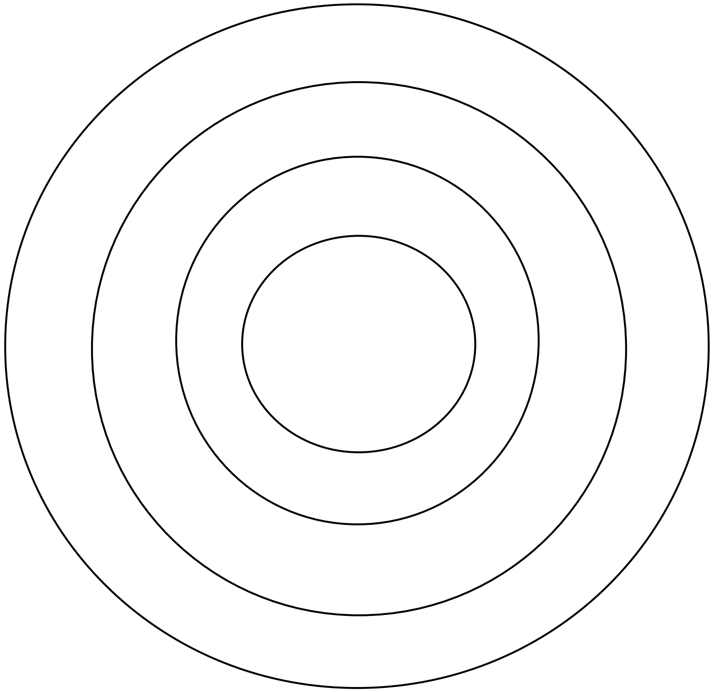
Queries for Checking Consistency (Model Validation)

- 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 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

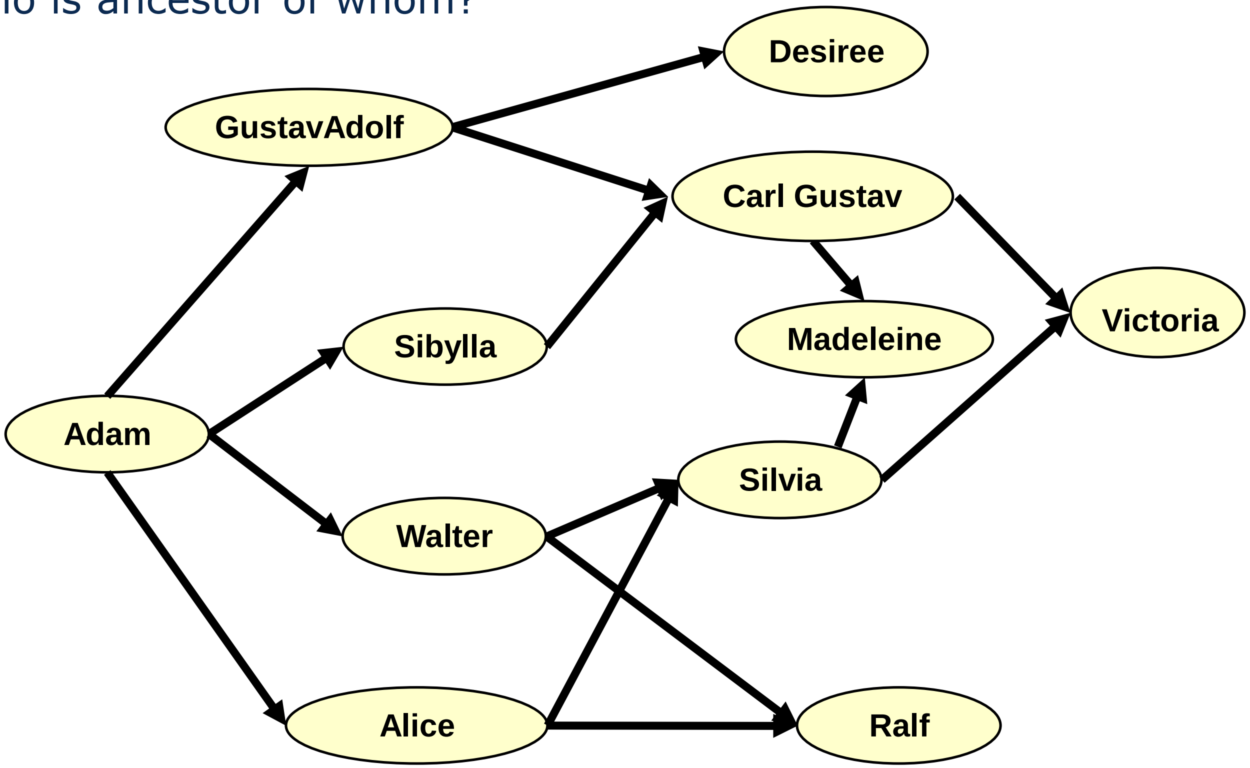
- With the “Same Generation” Problem
- How to query and search in a DAG
- How to layer a DAG – a simple structuring problem

- 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

- Given any acyclic relation, it can be made layered
  - Same Generation analysis creates layers for trees or DAGs
- Example: layering a family tree:
  - Who is whose contemporary?
  - Who is ancestor of whom?

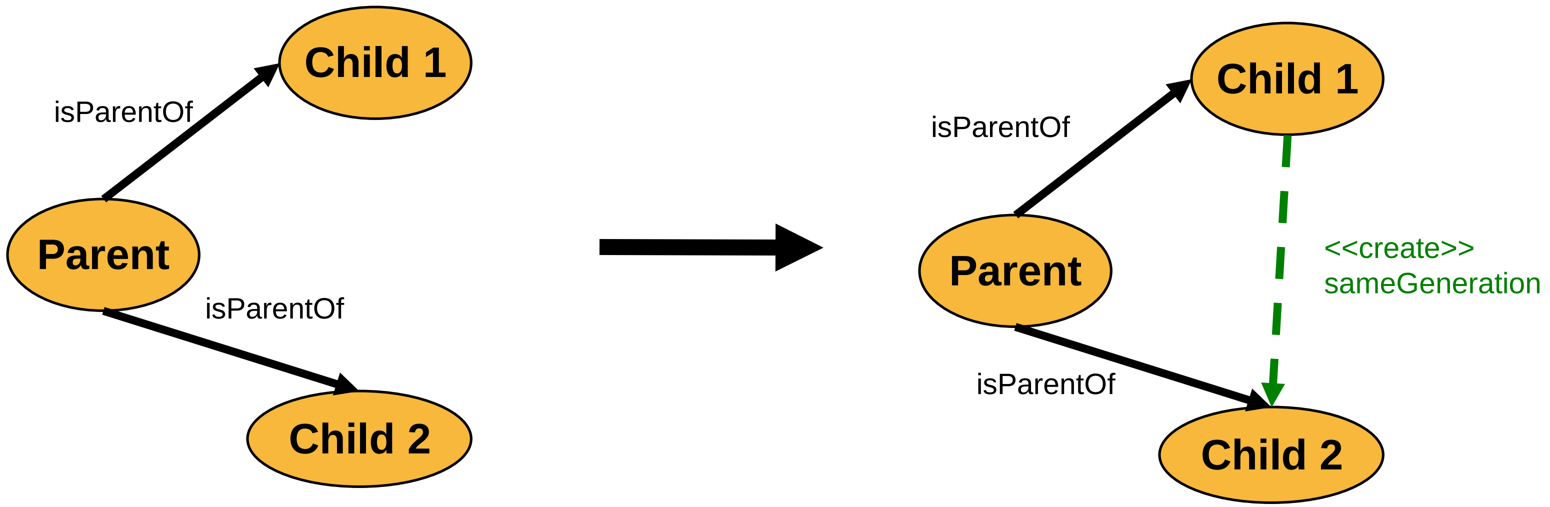


- Parenthood can be described by a *graph pattern*
- We can write the graph pattern also in logic:

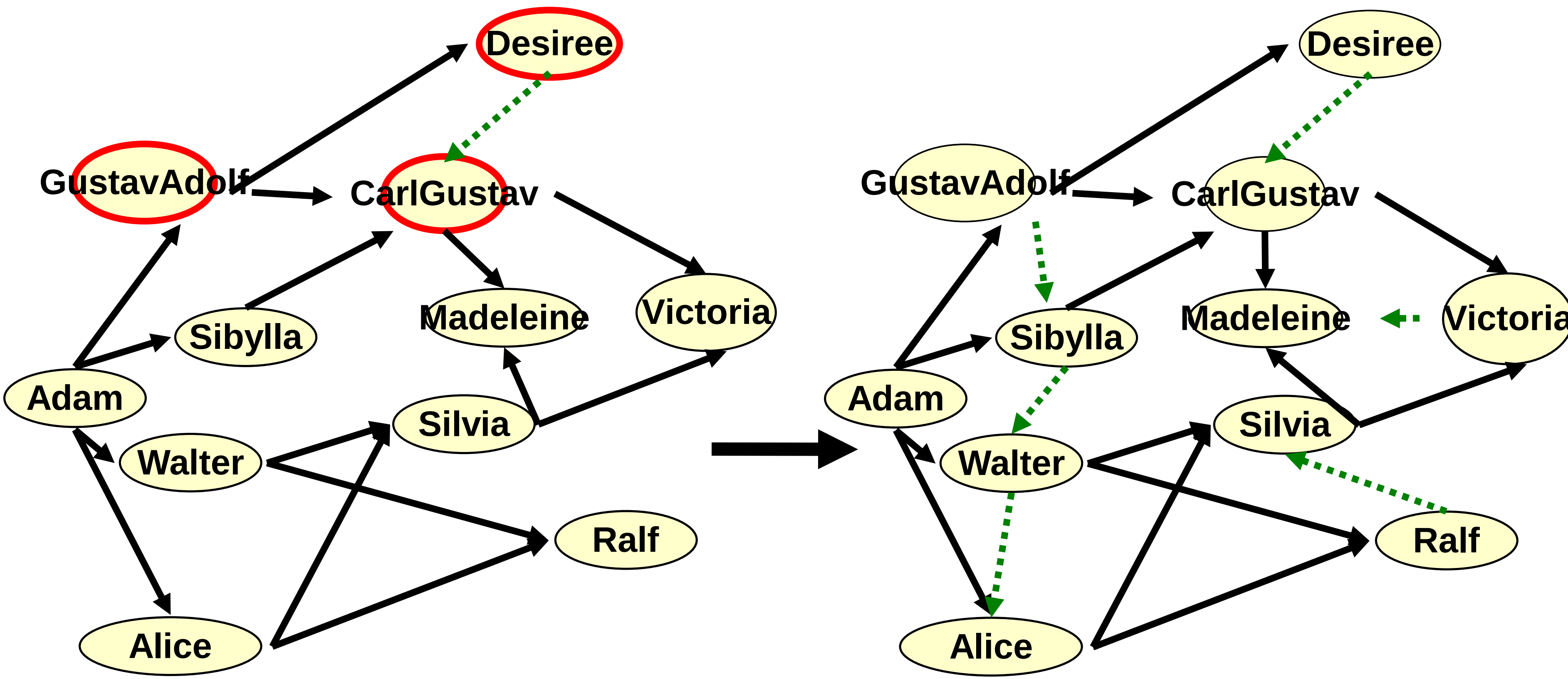
```
isParentOf(Parent,Child1) && isParentOf(Parent,Child2)
```

- And define the rule

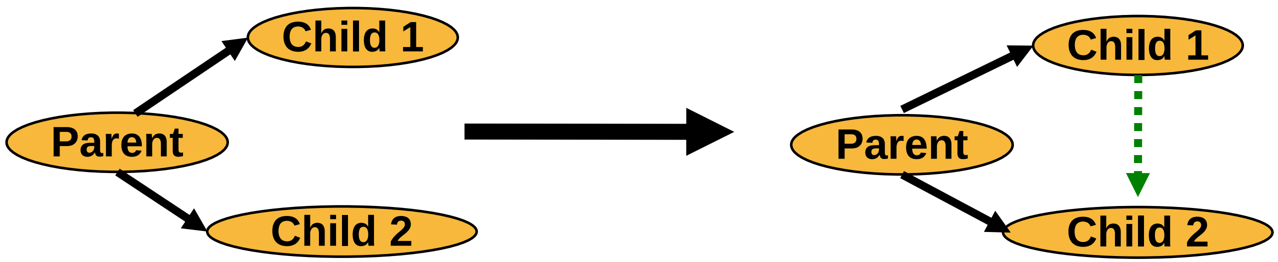
```
if isParentOf(Parent,Child1) && isParentOf(Parent,Child2)
then sameGeneration(Child1,Child2)
```



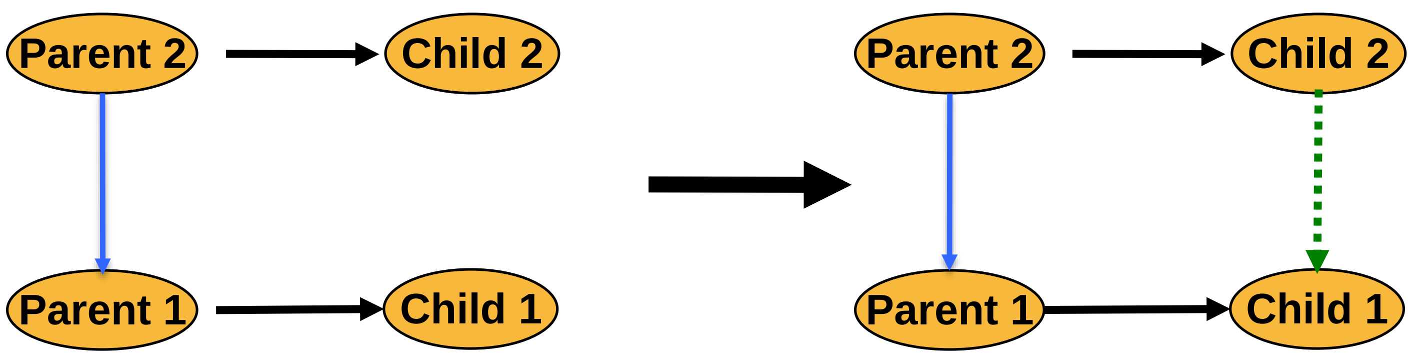
Impact of Rule on Family Graph



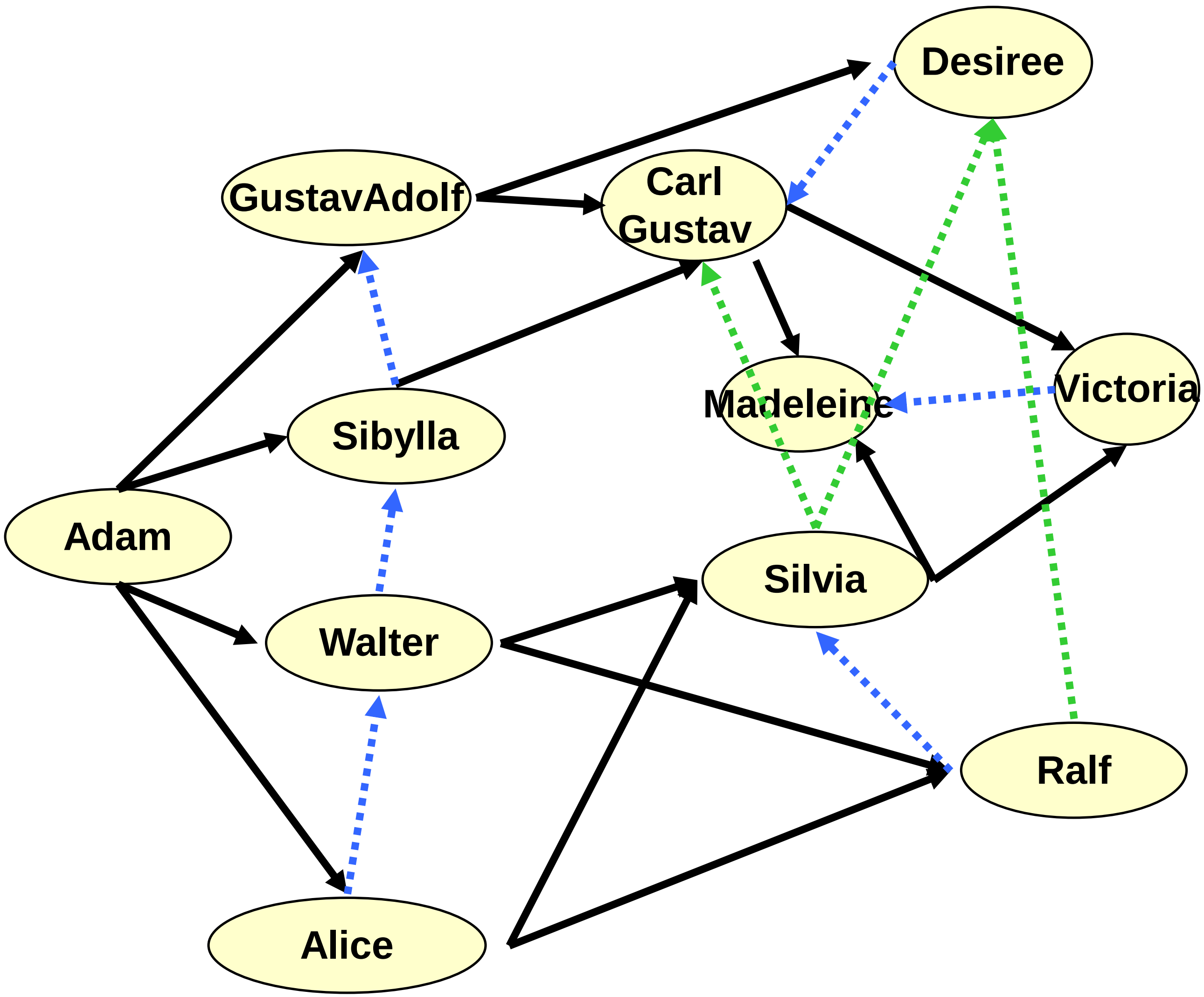
- Base rule: Beyond sisters and brothers we can link all people of same generation



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



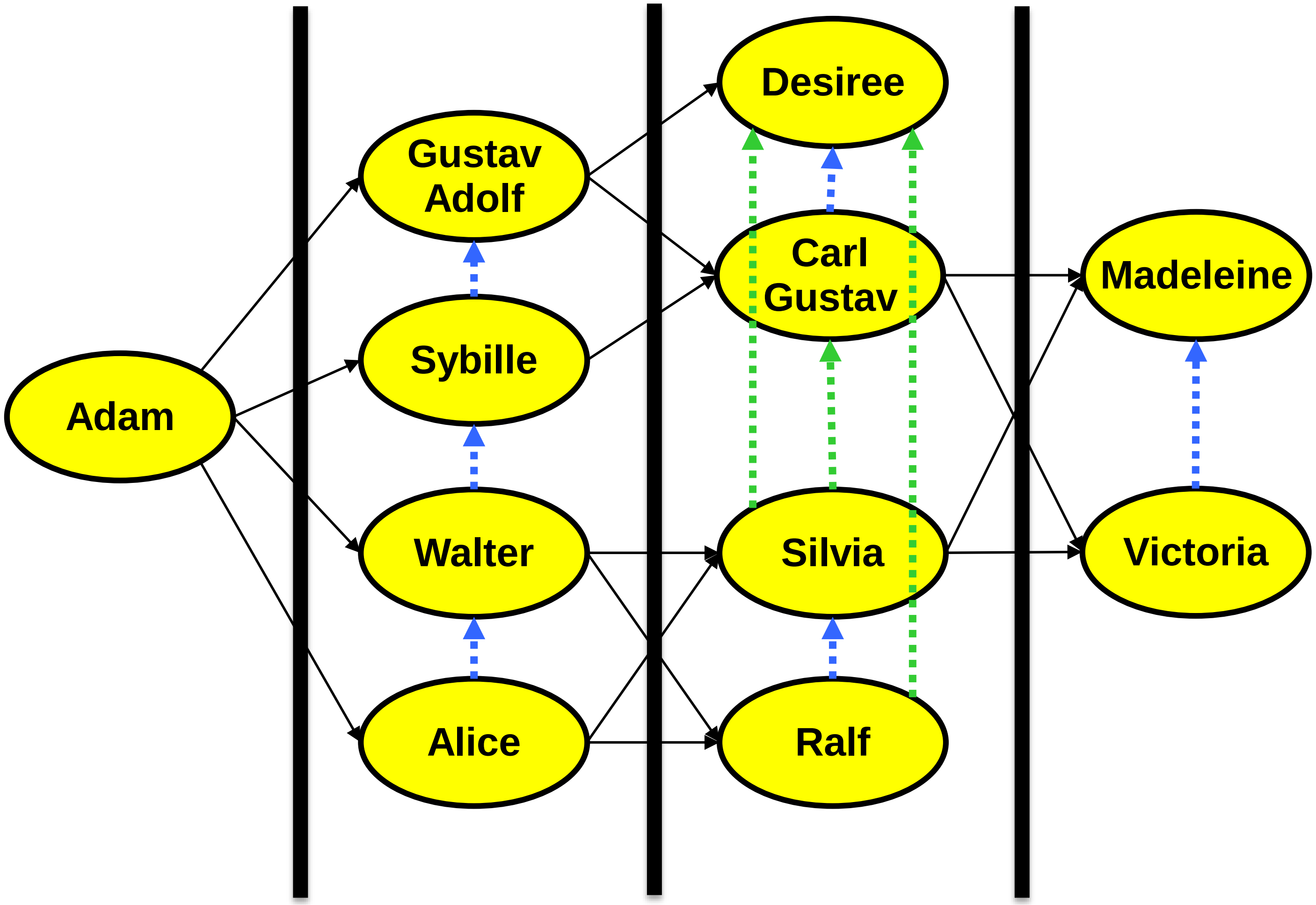
Impact of Transitive Rule



"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

The Generations as Layers



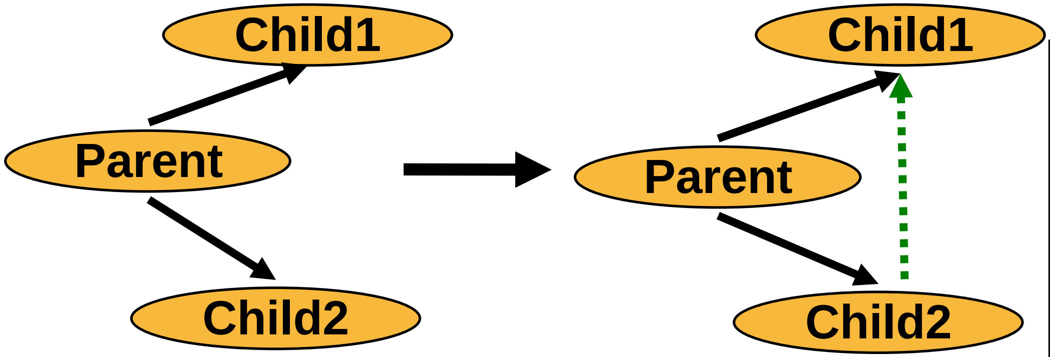
12.4.2 SEARCHING GRAPHS – SEARCHING IN SPECIFICATIONS WITH DATALOG AND EARS

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 / all orders of applying rules lead to the same result
    - **Recursion**: The system is recursive, since relation "Same Generation" is used and defined
    - **Termination**: terminates, if all possible edges are added, latest, when graph is complete
- EARS compute
  - Reachability of nodes
  - Paths in graphs
- "Same Generation" can be used for graph analysis

- Rule systems can be noted textually or graphically (DATALOG vs. 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).
```

- **S**ingle Source **M**ultiple Target **P**ath **P**roblem – SMPP
- **M**ultiple Source **S**ingle Target **P**ath **P**roblem – MSPP
- **M**ultiple Source **M**ultiple Target **P**ath **P**roblem – MMPP

```
# 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, ...}
```

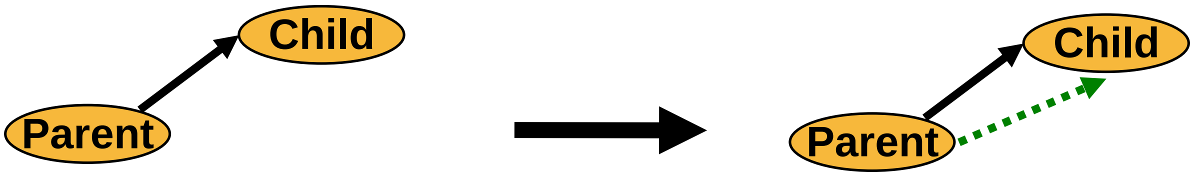
- The Swiss-Knife of Graph Analysis

12.5 REACHABILITY QUERIES WITH TRANSITIVE CLOSURE IN DATALOG AND EARS

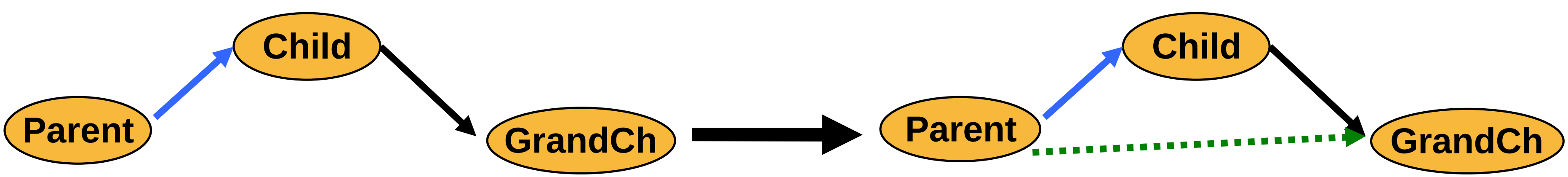
Who is Descendant of Whom?

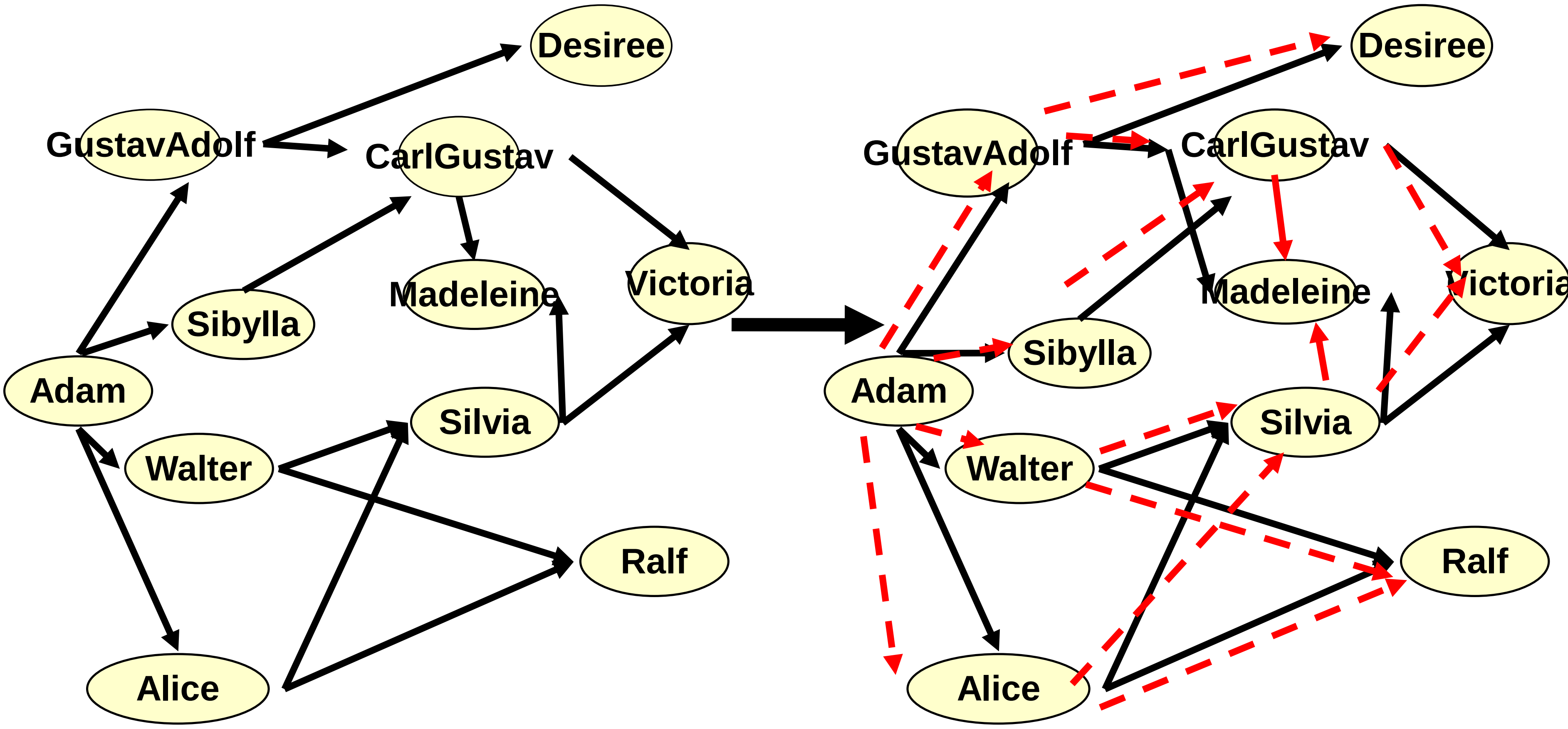
- Sometimes we need to know *transitive* edges, i.e., edges after edges of the same color
  - Question: what is *reachable* from a node?
  - Which descendants has Adam?
- Answer: Transitive closure calculates *reachability* over nodes
  - It contracts a graph, inserting masses of edges to all reachable 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)

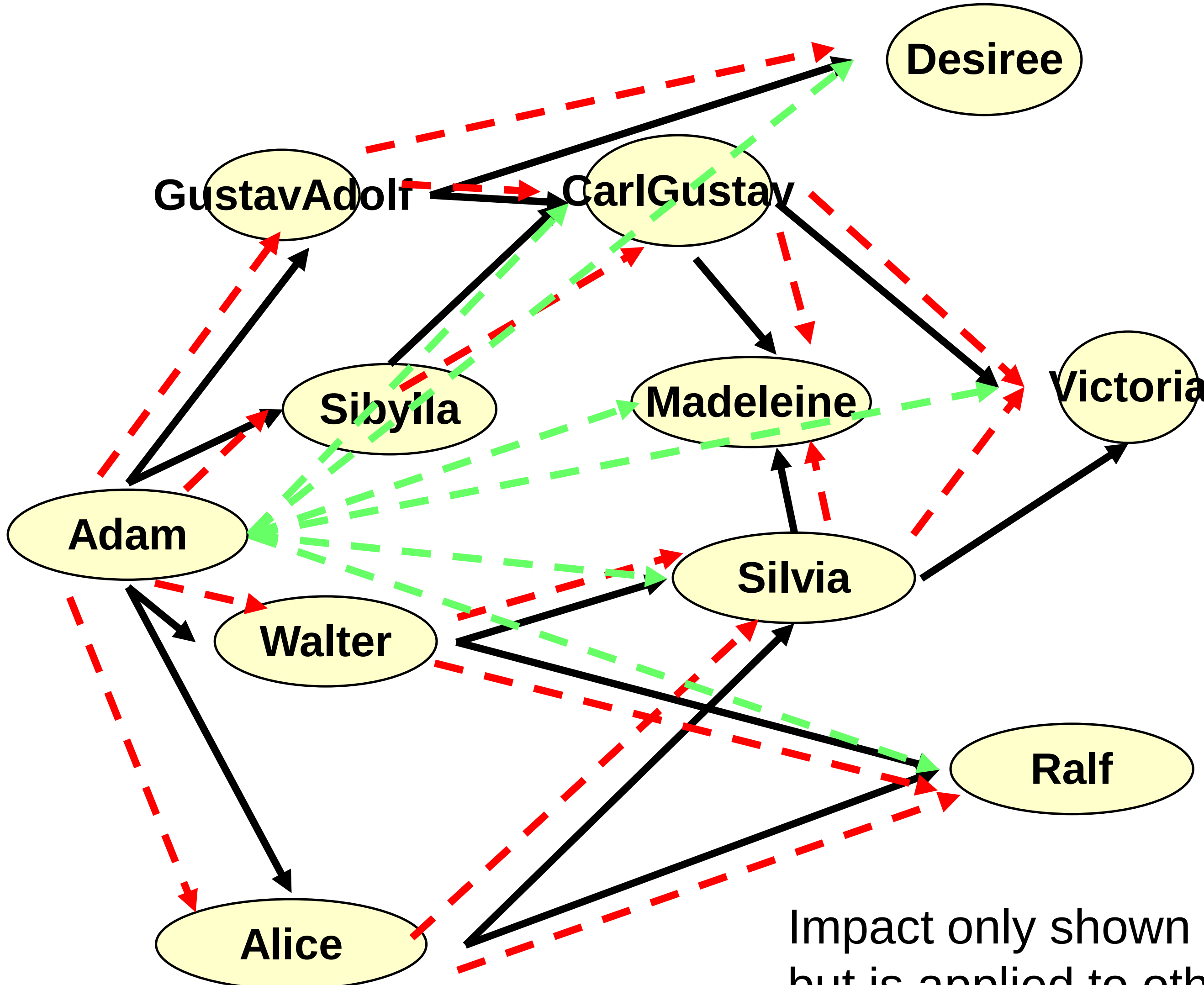
- Basic rule `descendant(Parent,Child) :- isChildOf(Parent,Child).`



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




Desiree
GustavAdolf
CarlGustav
Madeleine
Victoria
Sibylla
Adam
Silvia
Walter
Ralf
Alice
Desiree
GustavAdolf
CarlGustav
Madeleine
Victoria
Sibylla
Adam
Silvia
Walter
Ralf
Alice

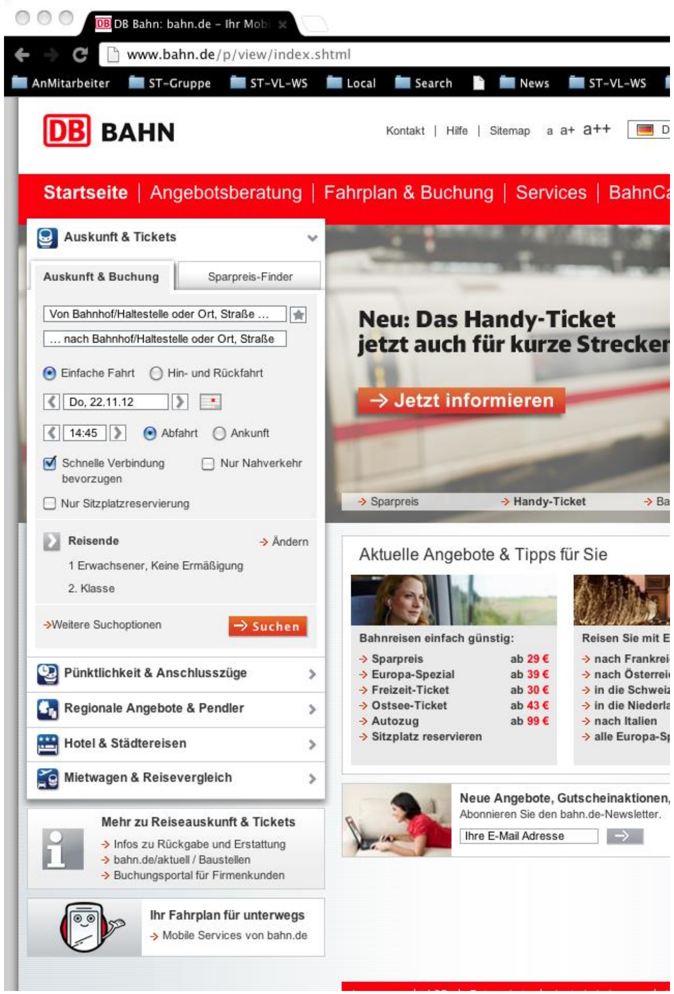


Impact only shown for Adam, but is applied to other nodes too

[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 neighboarship

- The info system of DB could be based on a graph of German railway stations.
- 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}`

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

Reading

- 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.

- 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

- 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 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

- 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

- 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

- Ebert, Jürgen; Riediger, Volker; Schwarz, Hannes; Bildhauer, Daniel **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.

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/

- 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.