Lecturer: Dr. Sebastian Götz

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

13. Validation of Graph-Based Models and Programs (Analysis and Consistency of Models)

Obligatory Reading

Balzert Kap. 1 (LE 2), Kap 2 (LE 4)

Maciaszek Chap 6-8

Goals

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 how to integrate them into tools for software quality assurance

Understand some basic concepts of simplicity in software models

Further 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/

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, 1-13. DOI= http://doi.acm.org/10.1145/1065167.1065169

Query Engines on Code and Models Using Logic

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 (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.

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/

Querying and Transformings Models with 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.

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 models and 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

13.1 Types of Graphs in Specifications

- Lists, Trees, Dags, Graphs
- Structural constrains on graphs
- (background information)

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)

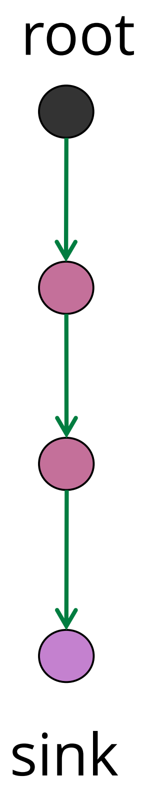
One sink

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)

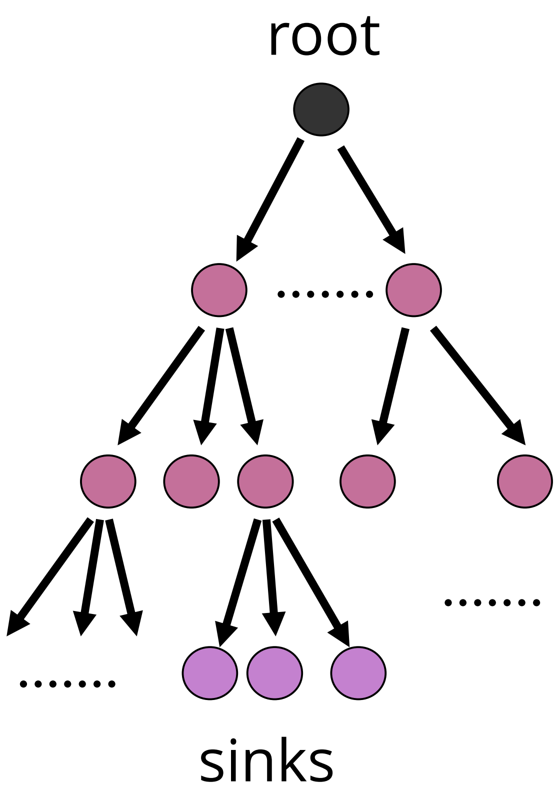
Every node has fan-in <= 1

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

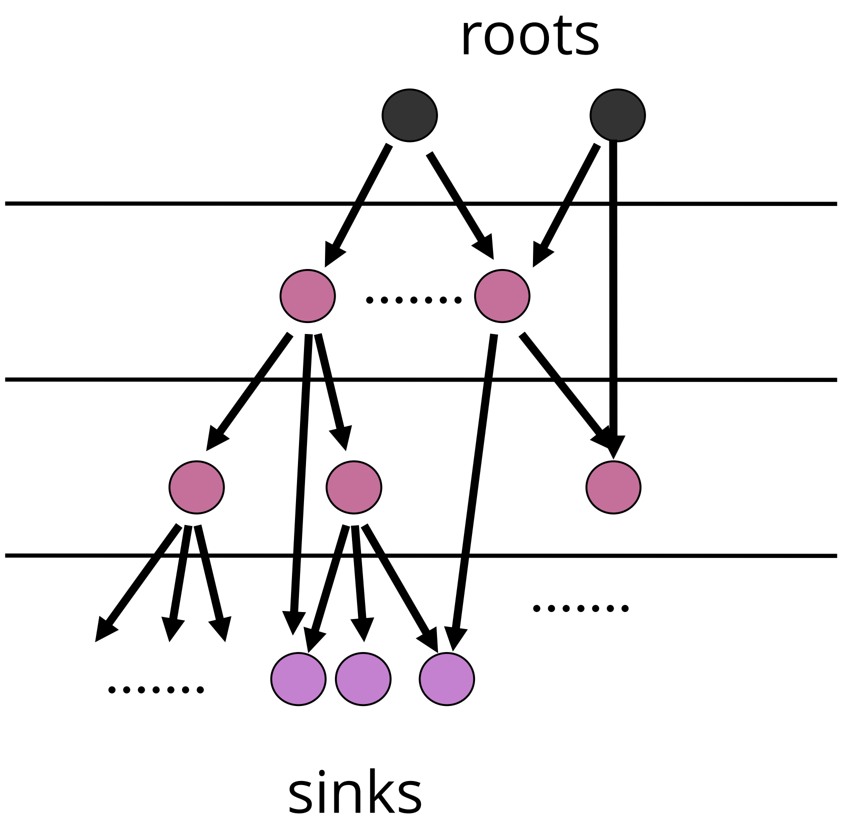
- Less constraints that in a total order

Weaker hierarchical abstraction feature

- Can be layered

Example

- UML inheritance dags
- Inheritance lattices



Link Trees
(Skeleton Trees with Overlay or Secondary Graphs)

A **Link Tree** is a skeleton tree with **overlay graph** (secondary links)

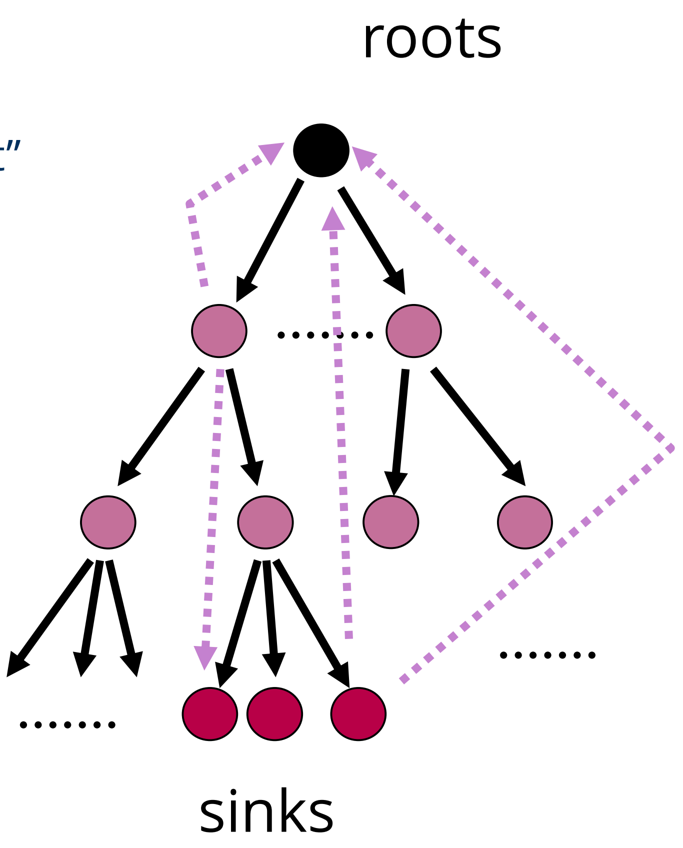
- ➢ Skeleton tree is primary
- ➢ Overlay graph is secondary: “less important”

Advantage of an Overlay Graph

- ➢ Tree can be used as a conceptual hierarchy
- ➢ References to other parts are possible

Example

- ➢ Link trees: Trees with links (references)
- ➢ 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

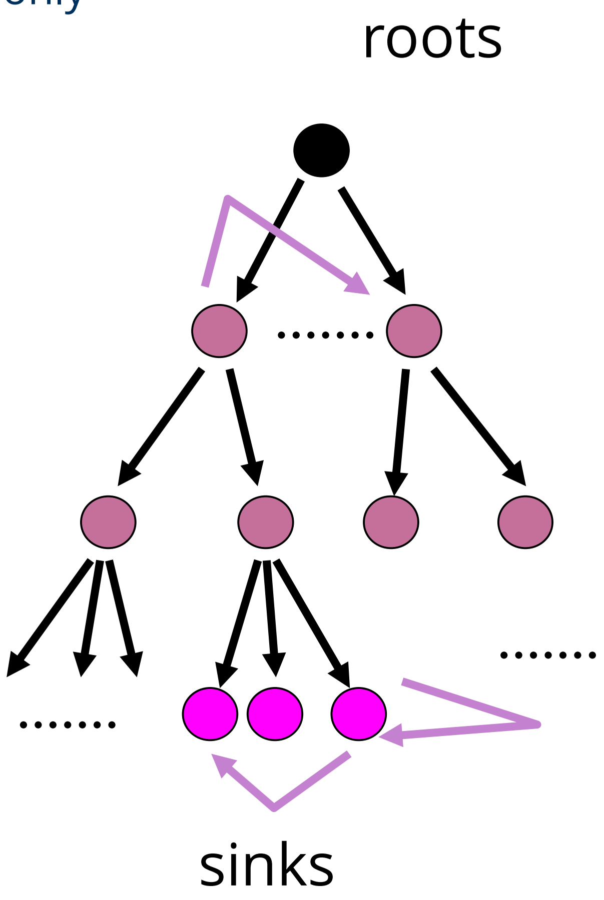
Graph can be “reduced” to one node

Advantage

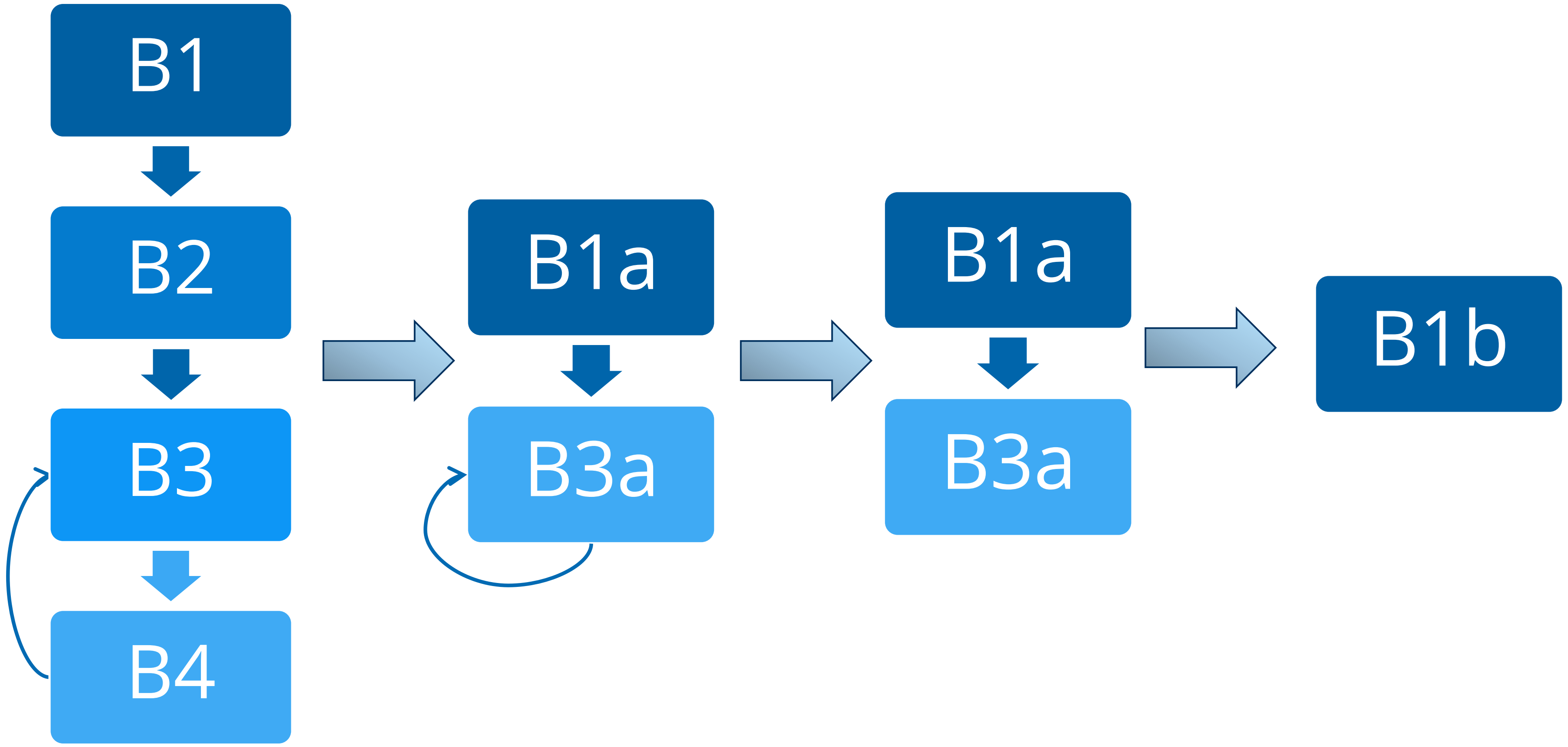
- ➢ Tree can be used as a conceptual hierarchy

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

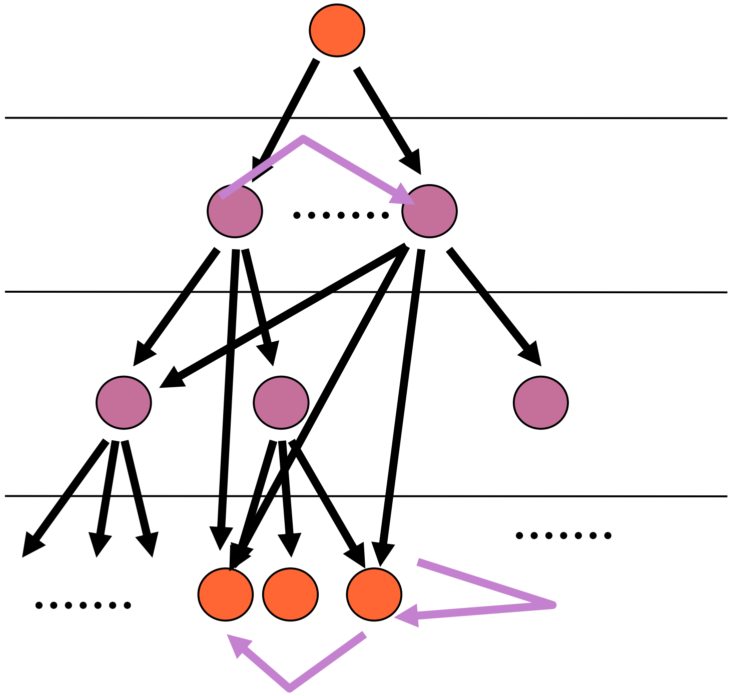
- ➢ Graph cannot be “reduced” to one node

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

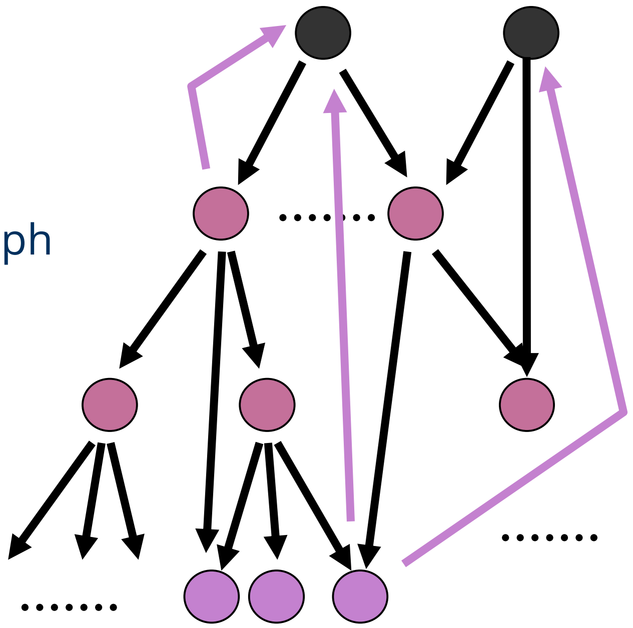
Many roots

- A digraph with one source is called flow graph

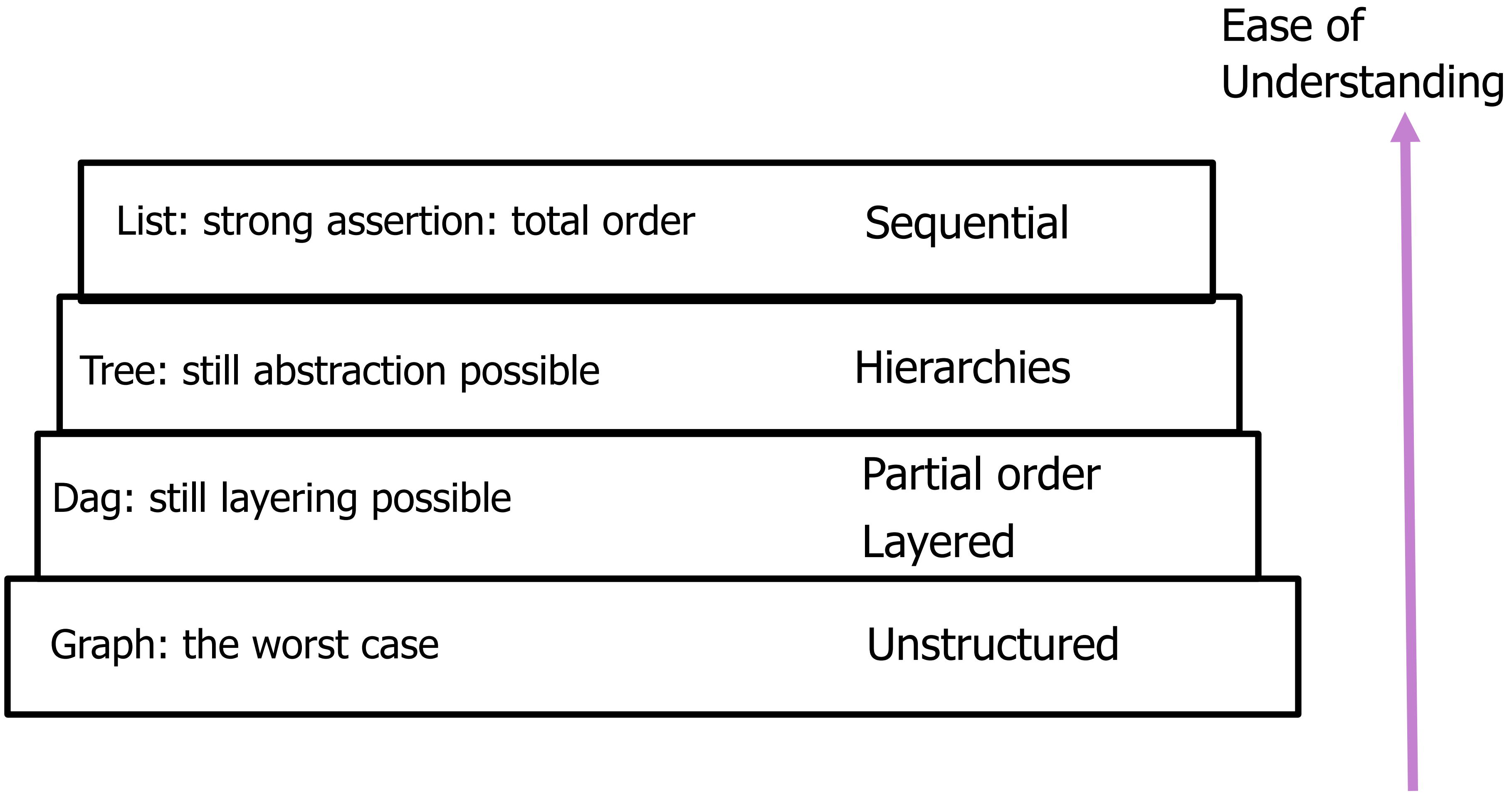
Many sinks

Example

- Many diagrammatic methods in Software Engineering
- UML class diagrams



Strength of Assertions in Graph-Based 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 Graph-Based 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
Link Tree: primary tree	Partial order Layered
Structured graph (reducible, skeleton dag)	Structured
Graph with analyzed features	Unstructured
Graph: the worst case	Unstructured

Ease of Understanding ↑

13.2 Methods and Tools for Analysis of Graph-Based Models

The Graph-Logic Isomorphism

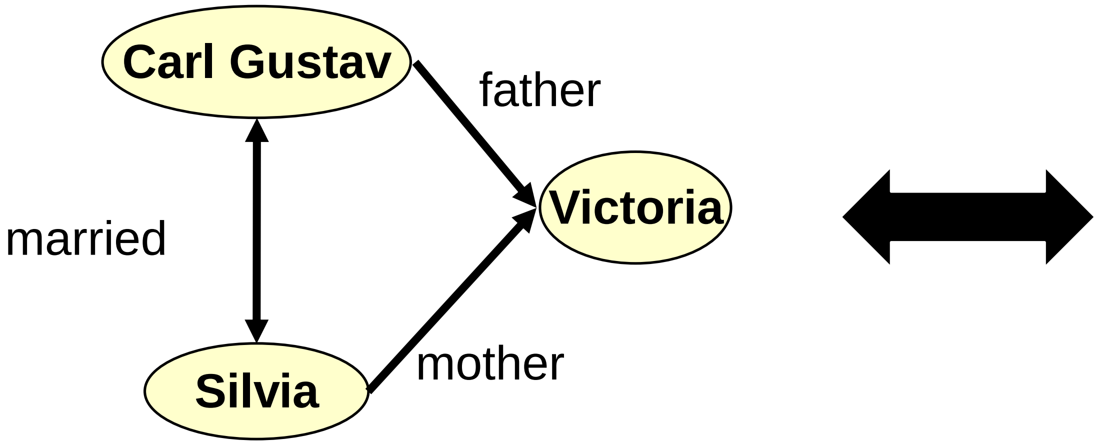
In the following, we will make use of the graph-logic isomorphism:

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



// 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

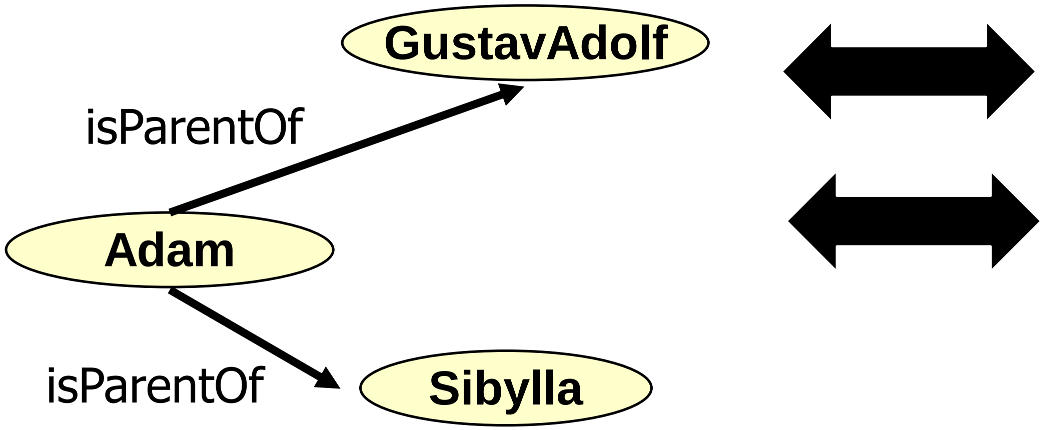
Graphs can also be noted textually

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).
```

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

13.2.1 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

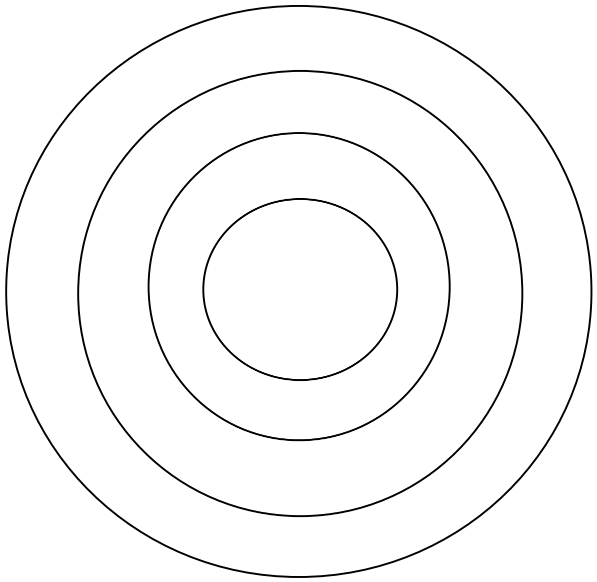
Layering of Systems

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 (-> ST-1)
- ➢ 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

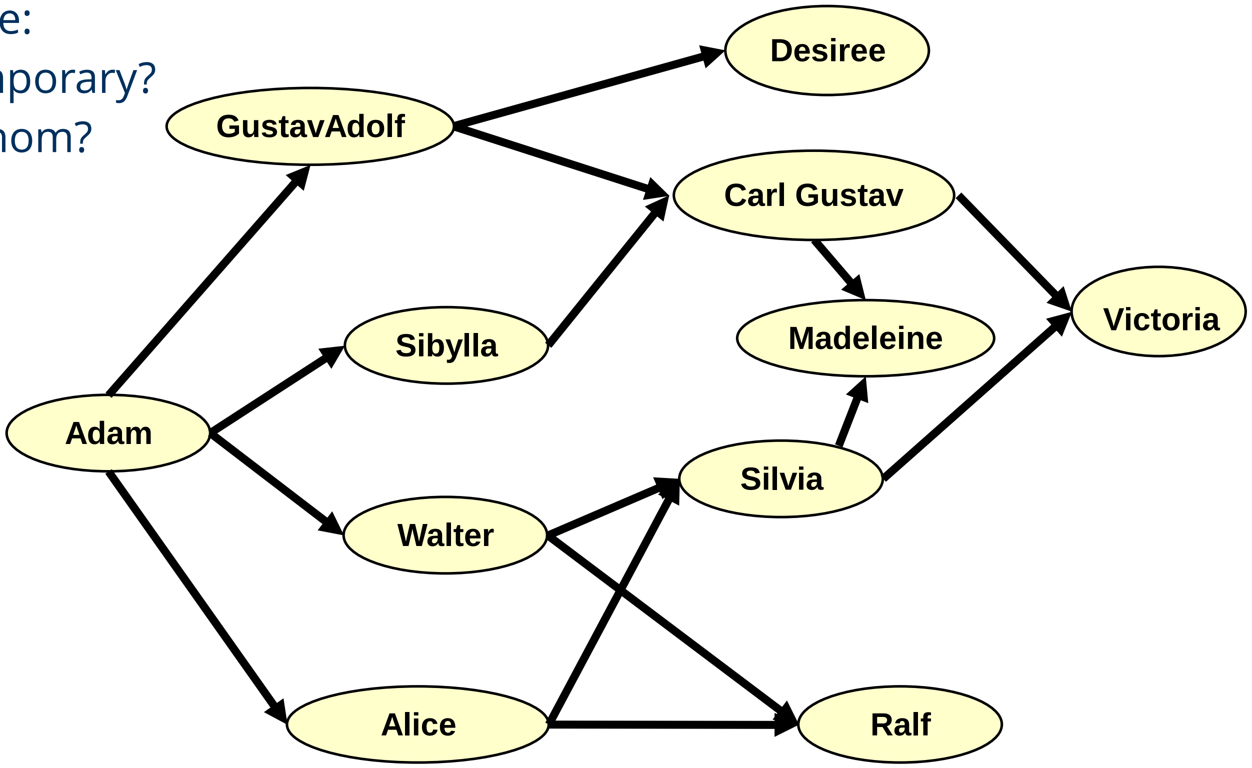
Layering of Acyclic Graphs

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

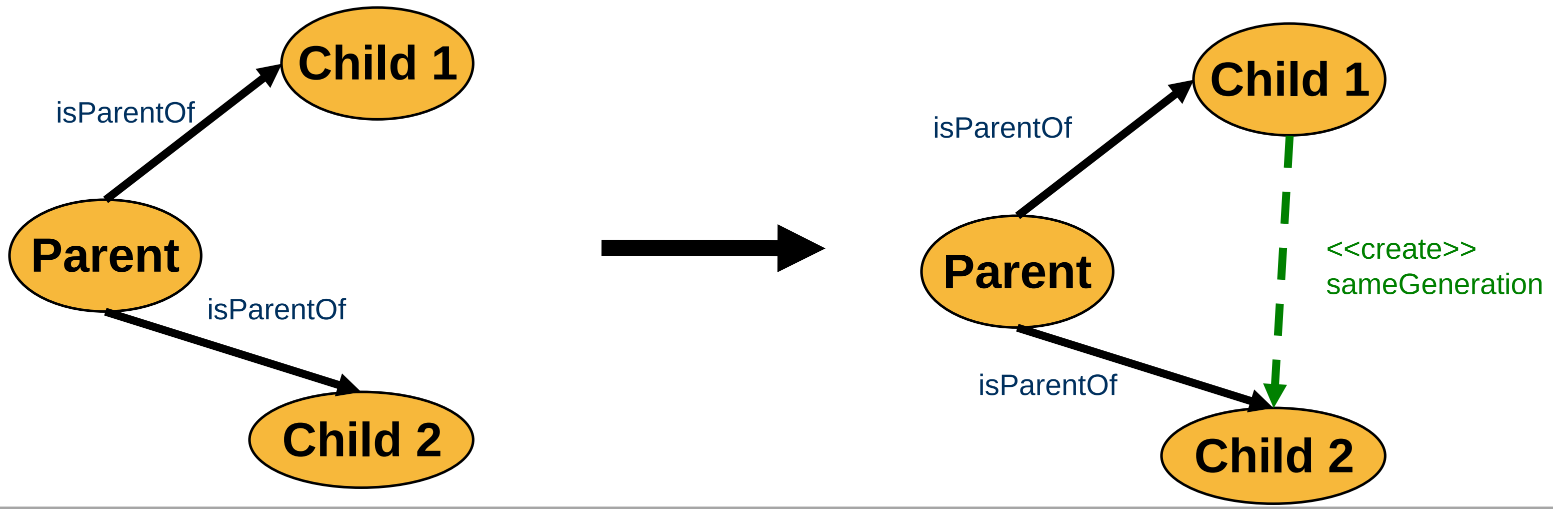
Parenthood can be described by a *graph pattern*

We can write the graph pattern also in logic:

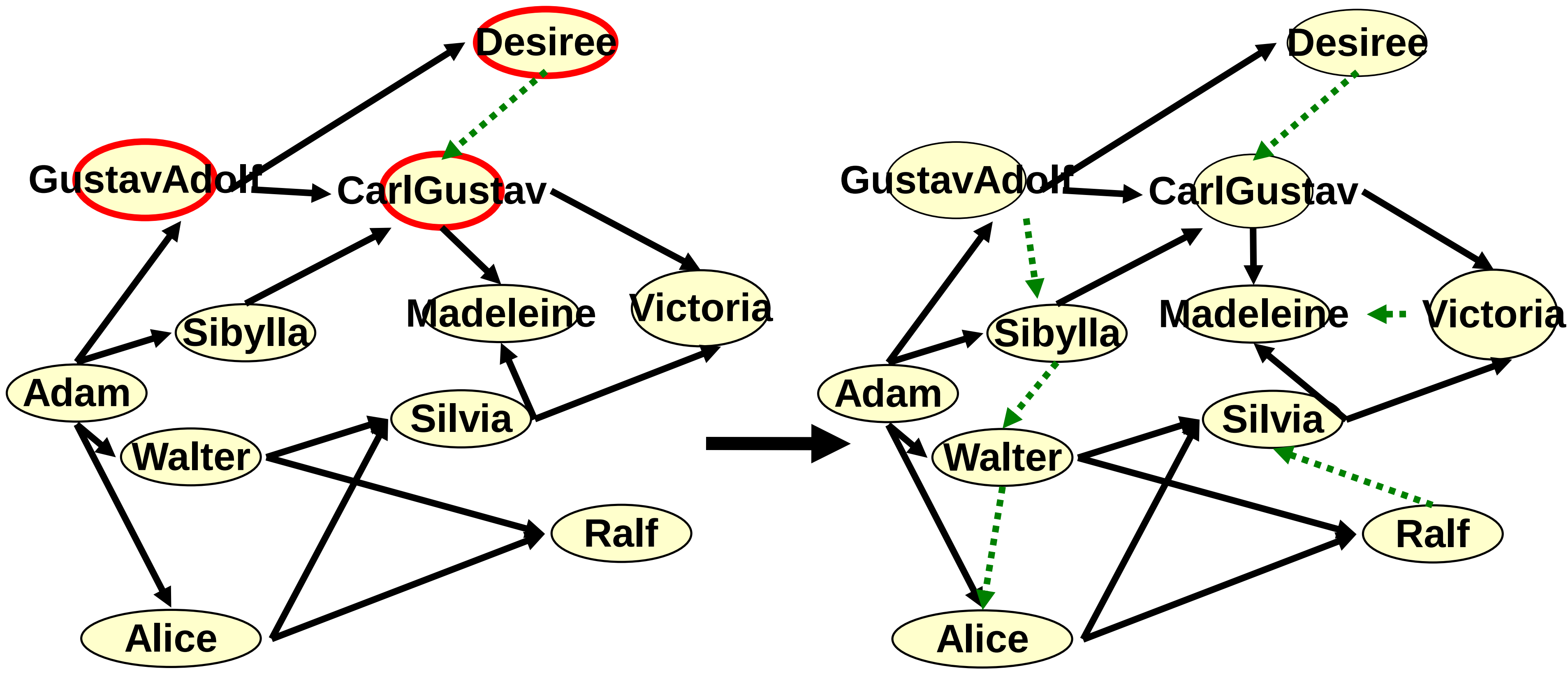
```
    isParentOf(Parent,Child1) && isParentOf(Parent,Child2)
```

And define the rule

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

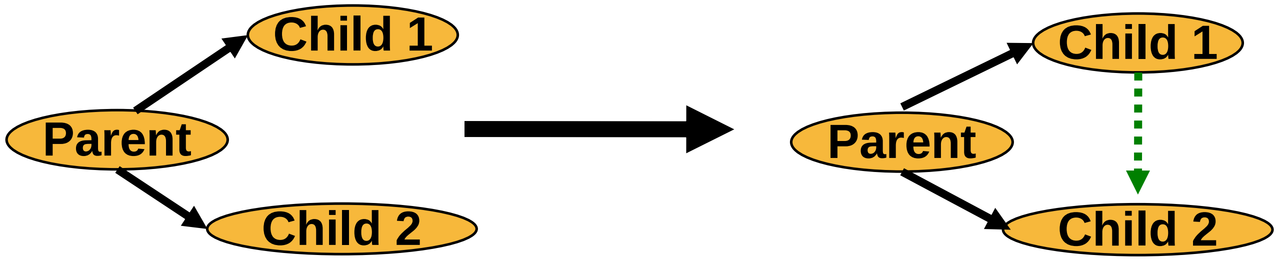


Impact of Rule on Family Graph

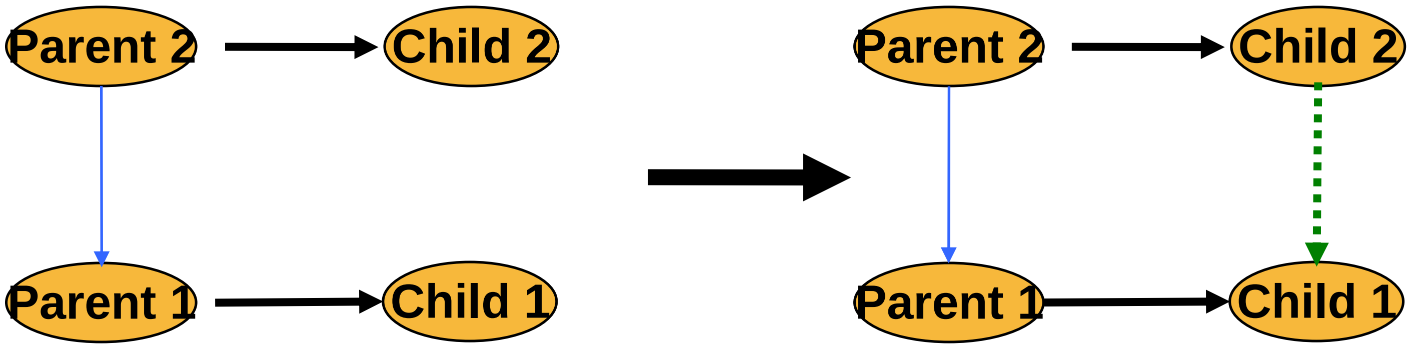


Rule set “Same Generation“

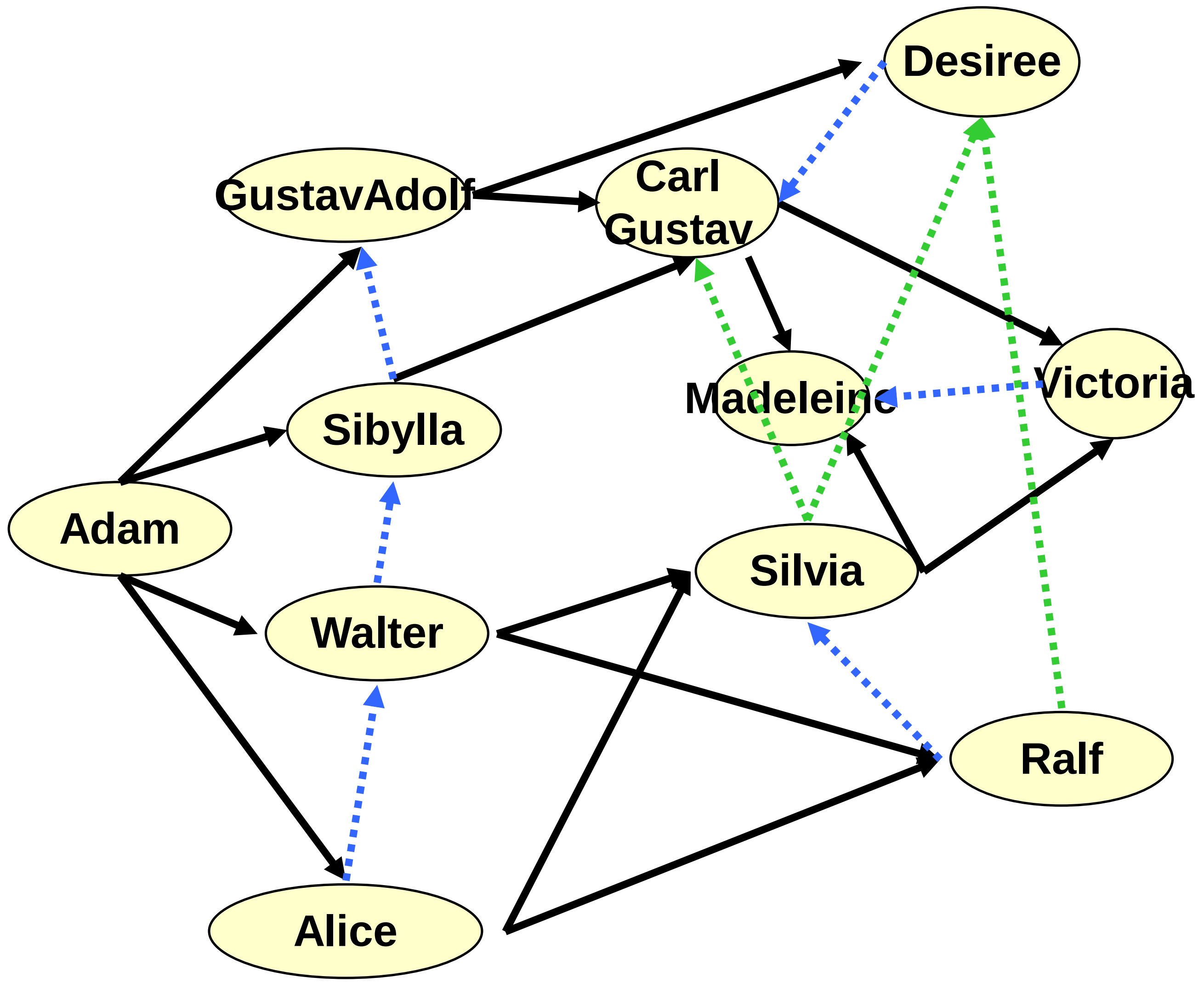
Base rule: Beyond sisters and brothers we can link all people of same generation



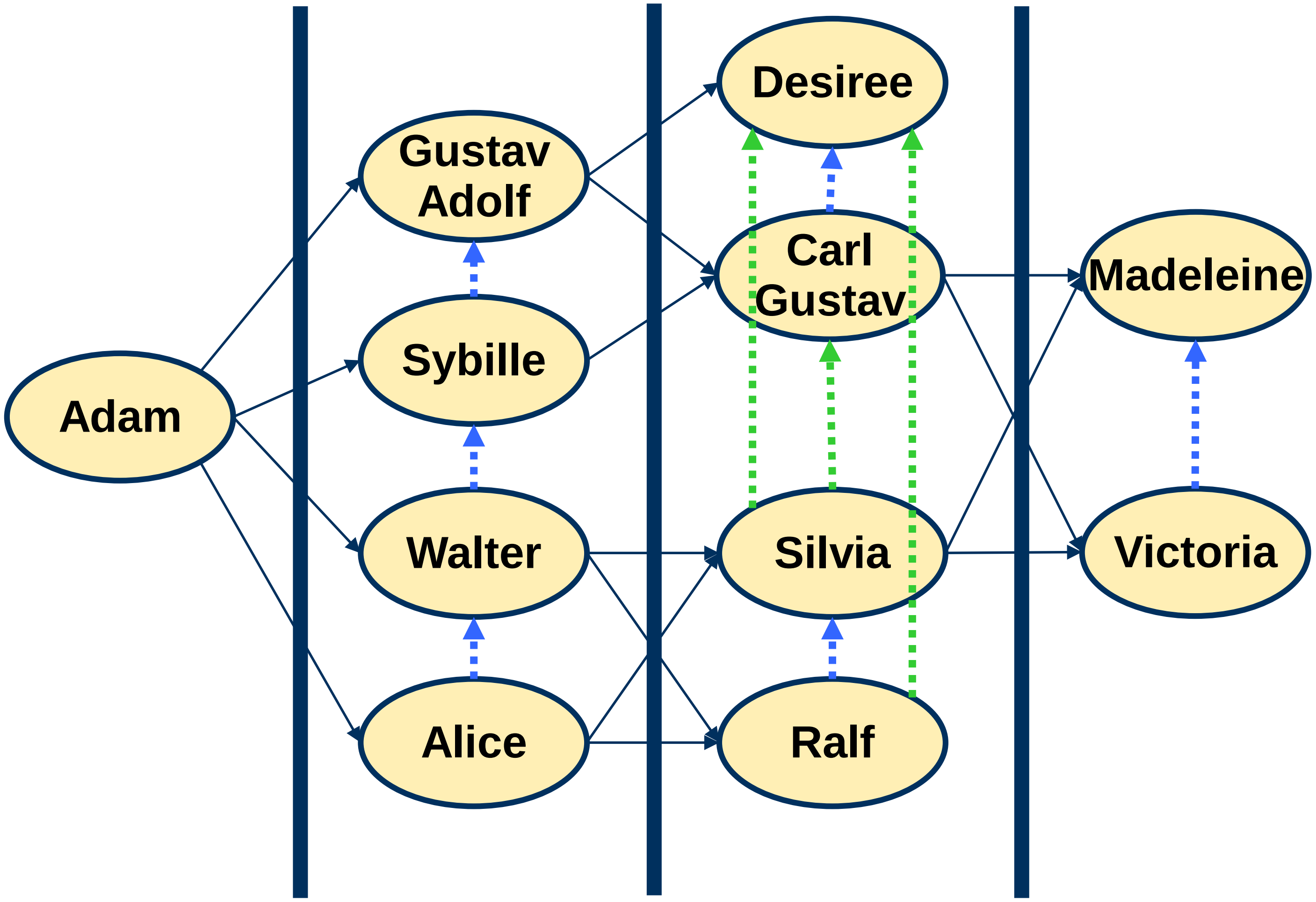
Additional rule (transitive): Enters new levels into the graph



Impact of Transitive Rule



The Generations as Layers



”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

13.2.2 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

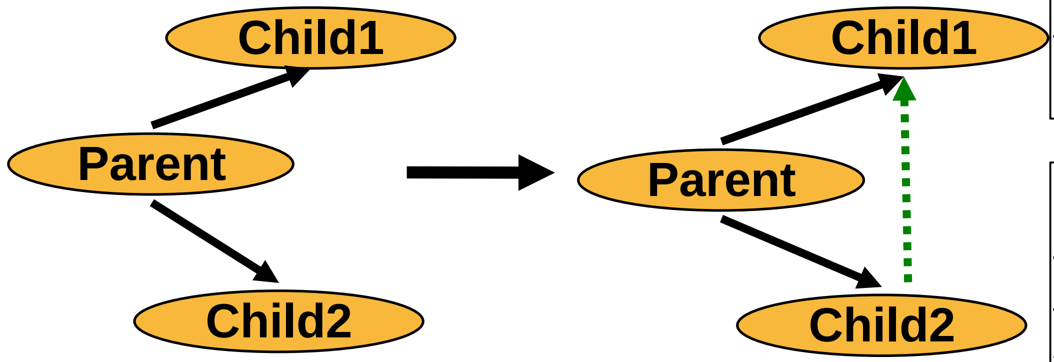
- **Termination**: terminates, if all possible edges are added, latest, when graph is complete
- EARS compute with graph query and graph analysis
  - Reachability of nodes
  - Paths in graphs
  - SameGeneration can be used for graph analysis

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

Single Source **M**ultiple Target **P**ath **P**roblem – SMPP

Multiple Source **S**ingle Target **P**ath **P**roblem – MSPP

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

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

# 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
```

13.3 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

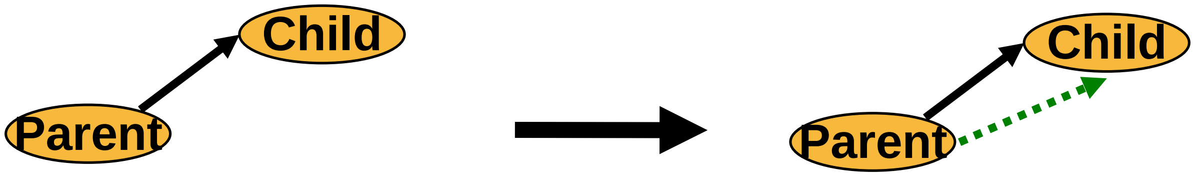
- ➢ It contracts a graph, inserting masses of edges to all reachable nodes
- ➢ It contracts all paths to single edges
- ➢ It makes reachability information explicit

After transitive closure, it can easily be decided whether a node is reachable or not

- ➢ Basic premise: base relation is *not changed* (offline problem)

Transitive Closure as Datalog Rule System or EARS

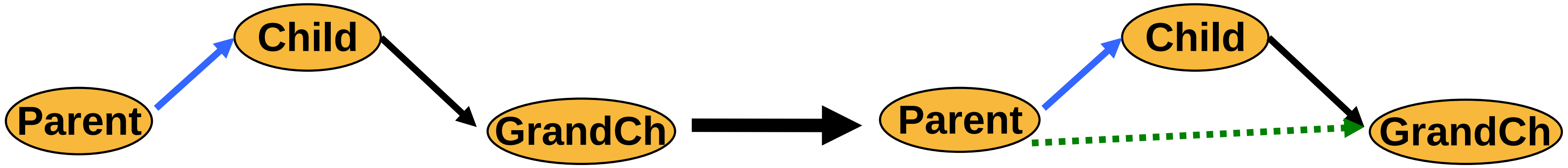
Basic rule `descendant(V,N) :- isChildOf(V,N).`



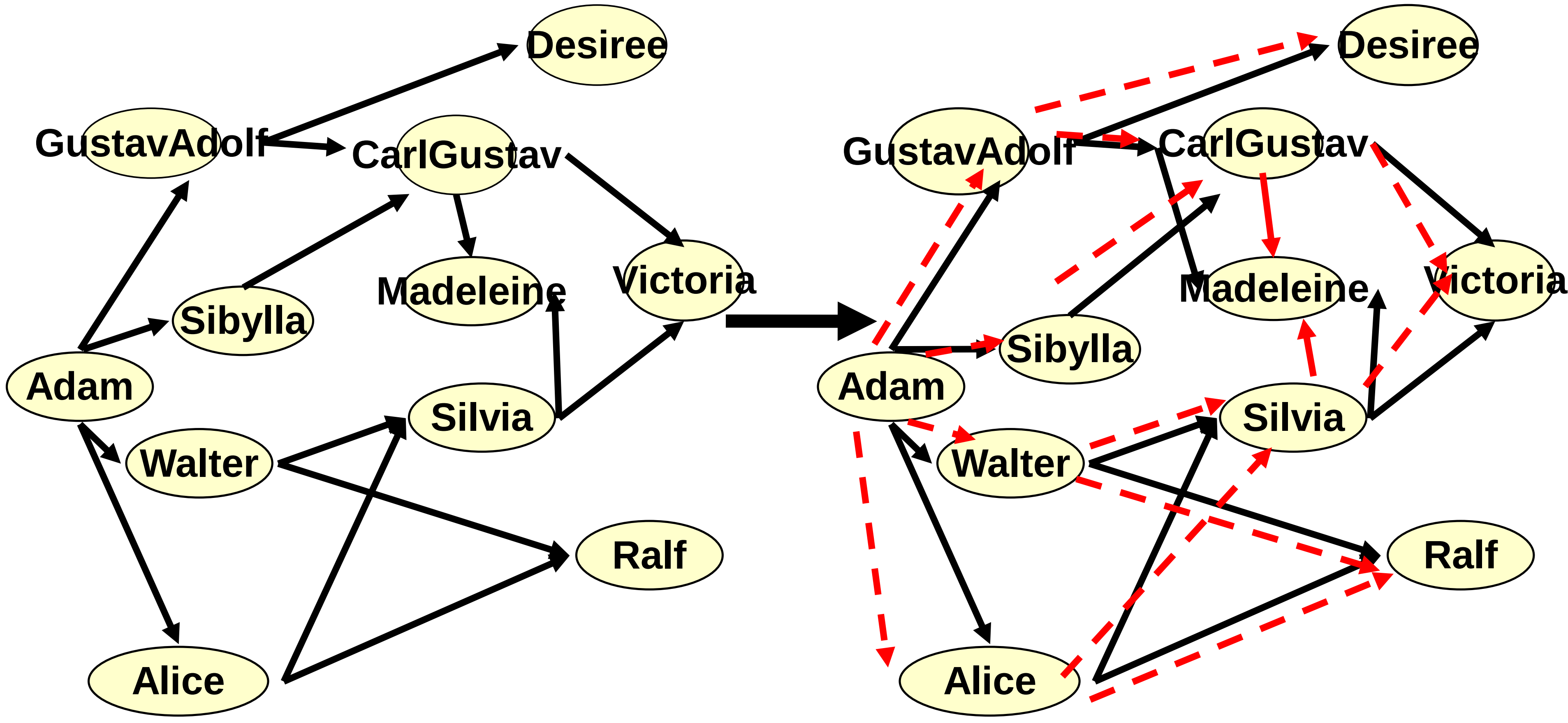
Transitive rule (recursion rule)

left recursive: `descendant(V,N) :- descendant(V,X),isChildOf(X,N).`

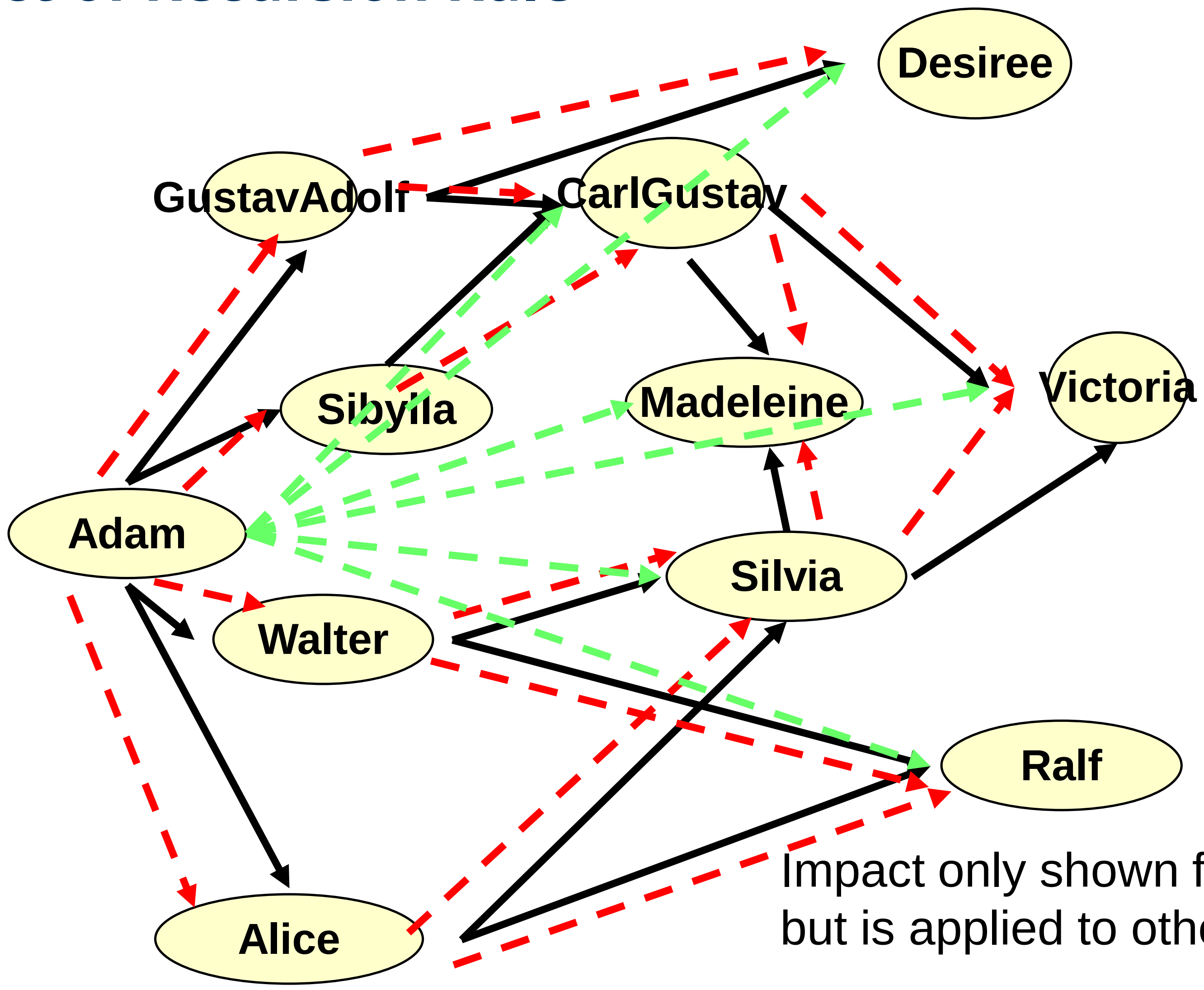
right recursive: `descendant(V,N) :- isChildOf(V,X), descendant(X,N).`



Impact of Basic Rule



Impact of Recursion Rule



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 neighborship

Exercise: Railway Routes as Reachability Queries

The info system of DB is based on a graph of German railway stations. If you query www.bahn.de, you end up in a Datalog query engine.

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)`**

DB Bahn: bahn.de – Ihr Mob
www.bahn.de/p/view/index.shtml
DB BAHN
Kontakt | Hilfe | Sitemap
Startseite | Angebotsberatung | Fahrplan & Buchung | Services | BahnCard | Urlaub
Meine Bahn | Login
Auskunft & Tickets
Auskunft & Buchung
Sparpreis-Finder
Von Bahnhof/Haltestelle oder Ort, Straße ...
... nach Bahnhof/Haltestelle oder Ort, Straße
Einfache Fahrt
Hin- und Rückfahrt
Do, 22.11.12
14:45
Abfahrt
Ankunft
Suchen
Neu: Das Handy-Ticket jetzt auch für kurze Strecken!
→ Jetzt informieren
Aktuelle Angebote & Tipps für Sie
Pünktlichkeit & Anschlusszüge
Regionale Angebote & Pendler
Hotel & Städtereisen
Mietwagen & Reisevergleich
Rund um die Buchung
Bahn-Vielfahrer
Gruppe, Familie & 55plus
Deutschland erleben!
Geschäftsreisen

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}`

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

The End