TECHNISCHE UNIVERSITÄT DRESDEN

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

16. How to Structure Large Models - Graph Structurings

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Gruppe Softwaretechnologie
http://st.inf.tu-dresden.de
Version 12-1.1, 05.12.12

Obligatory Reading

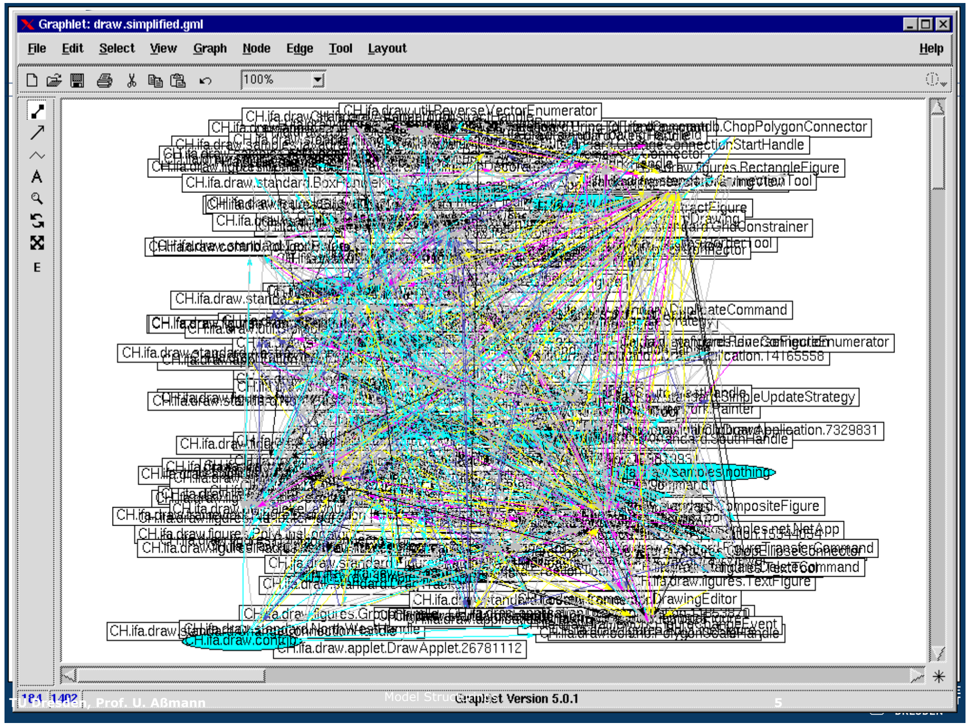
- Jazayeri Chap 3. If you have other books, read the lecture slides carefully and do the exercise sheets
- F. Klar, A. Königs, A. Schürr: "Model Transformation in the Large", Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, New York: ACM Press, 2007; ACM Digital Library Proceedings, 285-294. http://www.idt.mdh.se/esec-fse-2007/
- Tom Mens, Pieter Van Gorp. A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science 152 (2006) 125–142, doi:10.1016/j.entcs.2005.10.021
- T. Mens. On the Use of Graph Transformations for Model Refactorings. In GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/
- T. Fischer, Jörg Niere, L. Torunski, and Albert Zündorf, 'Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language', in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany (G. Engels and G. Rozenberg, eds.), LNCS 1764, pp. 296--309, Springer Verlag, November 1998. http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/1998/

Reducibility

- [Tarjan74] Robert E. Tarjan. Testing flow graph reducibility. Journal Computer System Science, 9:355-365, 1974.
- [ASU86] Alfred A. Aho, R. Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

Further Reading

- Reducible graphs
  - [ASU86] Alfred A. Aho, R. Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.
- Search for these keywords at
  - http://scholar.google.com
  - http://citeseer.ist.psu.edu
  - http://portal.acm.org/guide.cfm
  - http://ieeexplore.ieee.org/
  - http://www.gi-ev.de/wissenschaft/digitbibl/index.html
  - http://www.springer.com/computer?SGWID=1-146-0-0-0



The Problem: How to Master Large Models

- 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]

Problems

- Question: How to Treat the Models of a big Swiss Bank?
  - 25 Mio LOC
  - 170 terabyte databases
- Question: How to Treat the Models of a big Operating System?
  - 25 Mio LOC
  - thousands of variants
- Requirements for Modelling in Requirements and Design
  - We need automatic structuring methods
  - We need help in restructuring by hand...
- Motivations for structuring
  - Getting better overview
  - Comprehensibility
  - Validatability, Verifyability

??

Answer: Simon's Law of Complexity

- H. Simon. The Architecture of Complexity. Proc. American Philosophical Society 106 (1962), 467-482. Reprinted in:
- H. Simon, The Sciences of the Artificial. MIT Press. Cambridge, MA, 1969.

Hierarchical structure reduces complexity.
Herbert A. Simon, 1962

Remember, structuring is a basic engineering activity

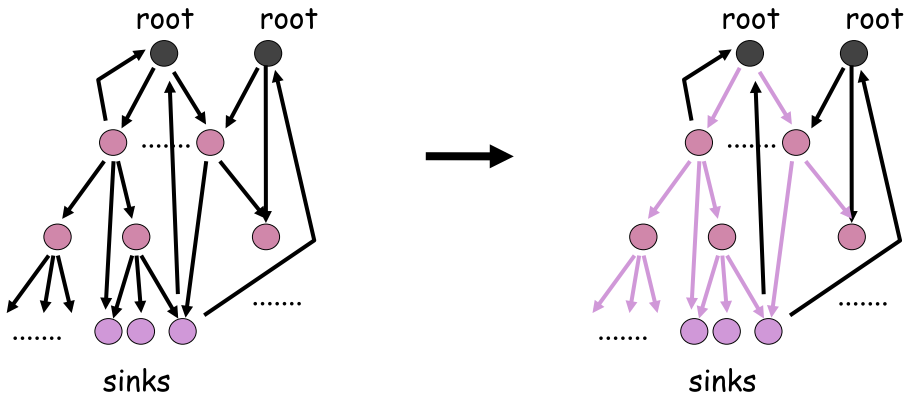
Model Transformations in General

- Model refactorings, lowerings, higherings, optimizers, and other transformations can be specified by graph transformations [Mens]
- Graph transformations can be specified by graph rewrite systems
  - Or by a programming language, of course

	Horizontal	Vertical
Endogeneous (within one language)	Structurings	Syntactic and semantic refinement
	Refactorings (course DPF)	
Exogeneous (crossing languages)	Language migration	PSM generation (see chapter MDA)
		PSI generation (code generation)

Idea of Structurings

- If a graph-based model is too complex, try structurings
- Structurings *overlay* graphs with *skeleton lists, trees, and dags*
- Structuring can be achieved with graph analysis, logic-based analysis, and graph rewriting
- Example: finding a spanning tree:



Idea of Structurings

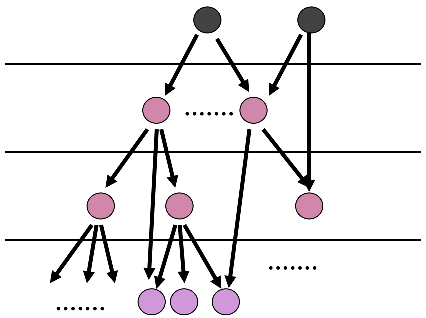
- Then, following the structure,
  - Sequential algorithms can be applied
  - Recursive algorithm schemas can be applied
  - Wavefronts can be applied
- Structures are nice for thinking and abstraction (see Simon's law)
  - In particular in analysis and design
- Structurings prepare further refactorings
  - The structural information can be exploited to further transform the code and to prove preservation of semantics
- Structurings need
  - Logics with types (e.g., F-Datalog)
  - Graph reachability analysis
  - Graph transformation

Overlaying a list on a dag

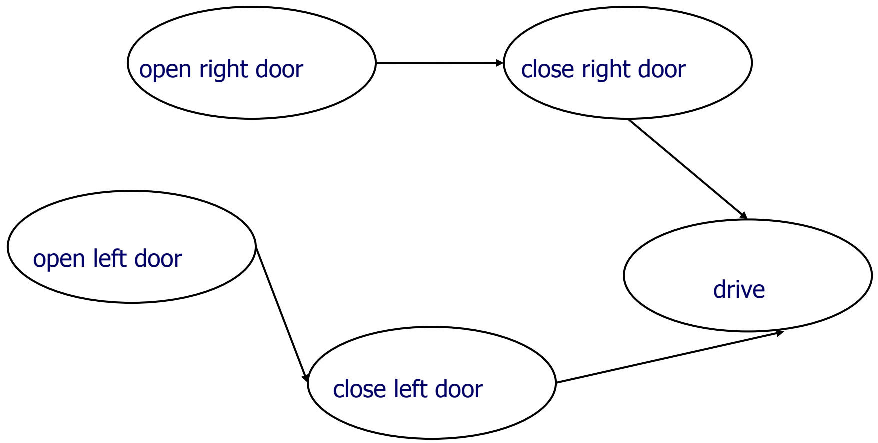
16.1 TOPOLOGIC SORTING OF DAGS (LAYERING)

Topologic Sorting on Dags

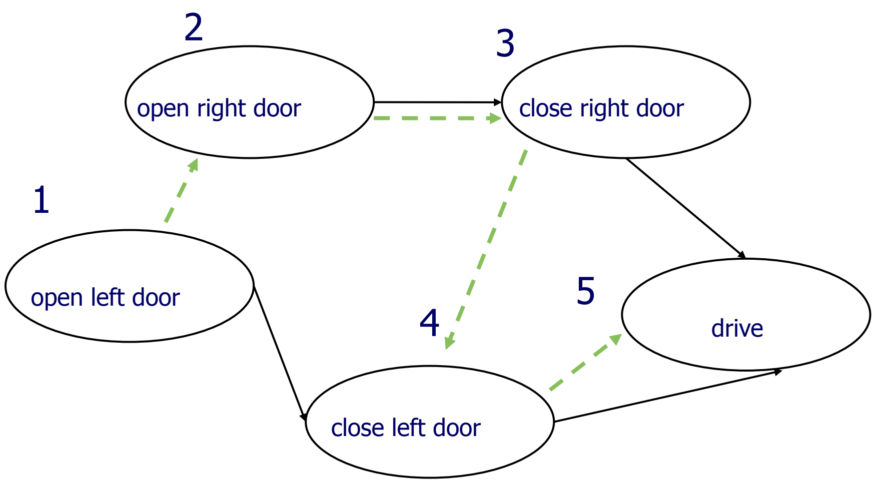
- If constraints for the partial order of some things are given, but no total order
- It doesn't matter in which order some things are executed
  - May be even in parallel
- There are many "legal" orderings, the topological sortings (topsorts, Totalordnung)



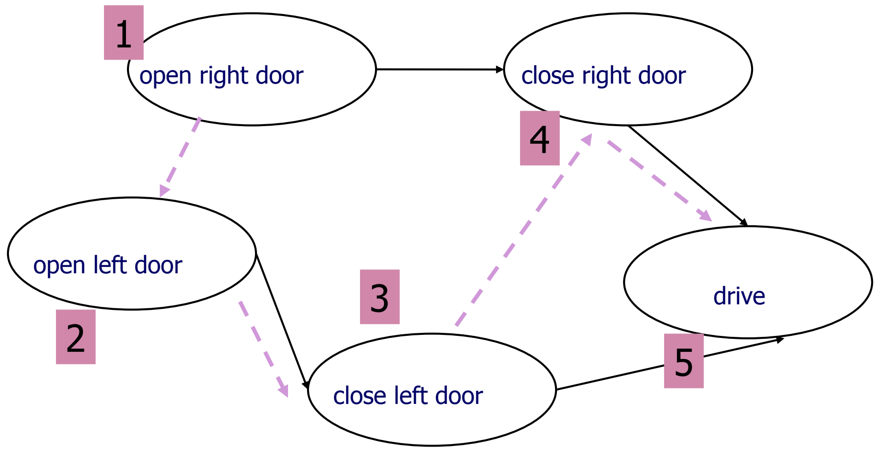
Partial Order for Car Departure



Topological Sorts on Car Departure



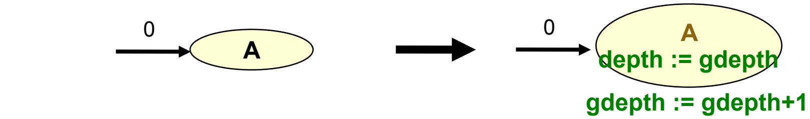
Topological Sorts on Car Departure



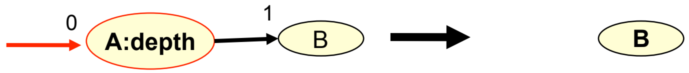
Topological Sort

- Topological sorting sorts the nodes with the „least many ancestors" first
- TopSort can be described by a subtractive graph rewrite system (SGRS)

TopSort-R1: Numbering entry nodes

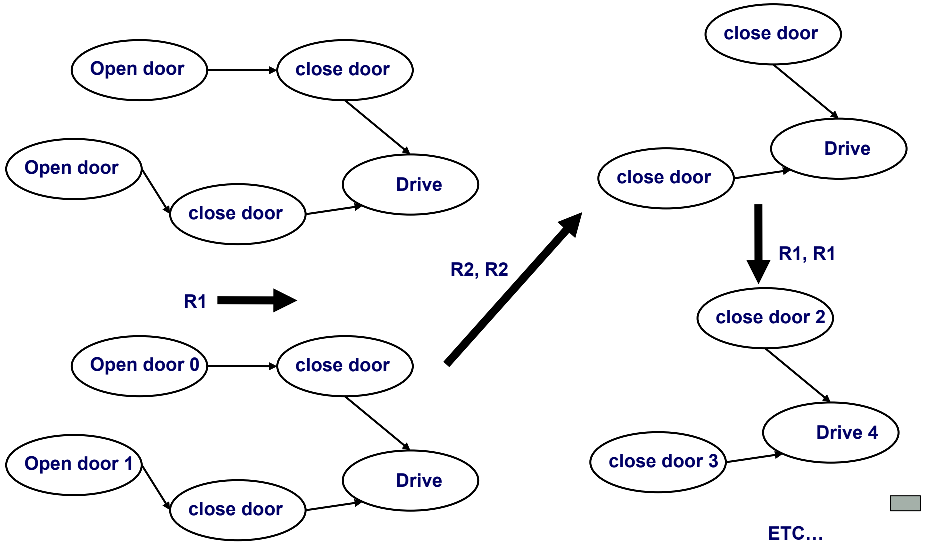


TopSort-R2: Contraction: Remove entry nodes



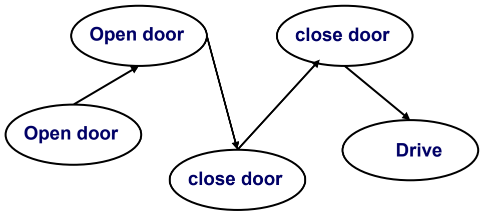
http://de.wikipedia.org/wiki/Topologische_Sortierung

Topological Sorts on Car Departure



Results: Topological Sortings

- The derivations of the GRS TopSort result in different topological sortings of the dag.
- For instance:

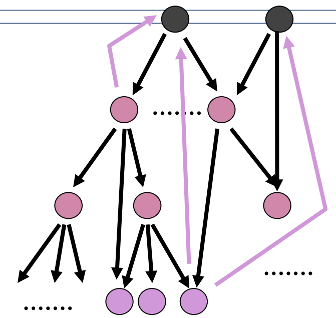


Benefit of TopSorts

- TopSorted dags are simpler
  - Because they structure partial orderings
  - Removing parallelism and indeterminism
- Question: why are all cooking recipes sequential?

Applications of TopSort

- Marshalling (serialization) of data structures
  - Compute a topsort and flatten all objects in the order of the topsort
- Package trees
  - Systems with big package trees can be topsorted and then handled in this order for differenzing between versions (regression tests)
- Task scheduling
  - Find sequential execution order for parallel (partially ordered) activities
- UML activity diagrams
  - Finding a sequential execution order
- Execution of parallel processes (sequentialization of a parallel application)
  - Execute the processes according to dependencies of a topsort
- Project management:
  - Task scheduling for task graphs (milestone plans): who does when what?
  - Find a topsort for the construction of your next house!



How to make an arbitrary relationship acyclic: overlaying a graph with a dag

16.2 STRONGLY CONNECTED COMPONENTS

Strongly Connected Components (Acyclic Condensation)

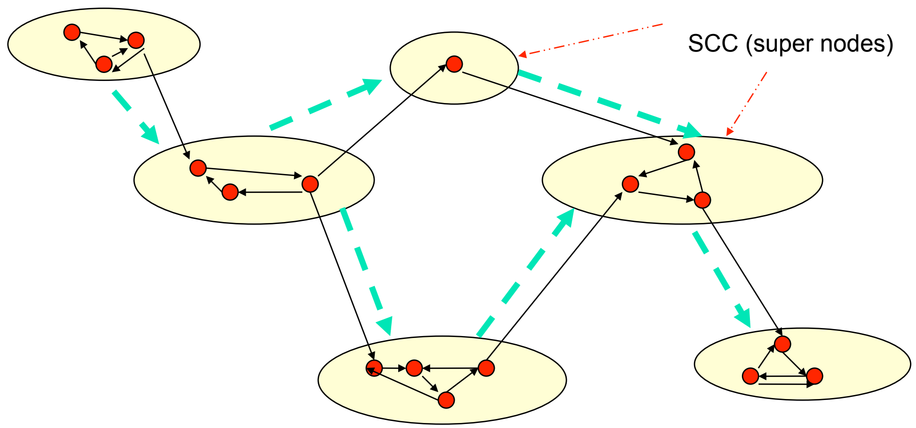
- The acyclic condensation asks for mutual reachability of nodes, hence for the effect of cycles in graphs
- A digraph is strongly connected, if every node is reachable from another one
- A subgraph of a graph is a strongly connected component (SCC)
  - If every of its nodes is strongly connected
- The reachability relation is symmetric
  - All edges on a cycle belong to the same SCC
- How to compute reachability:
  - Declaratively: Specification with an EARS or recursive Datalog:

```
sameSCC(X,Y) :- reachable(X,Y), reachable(Y,X).
```

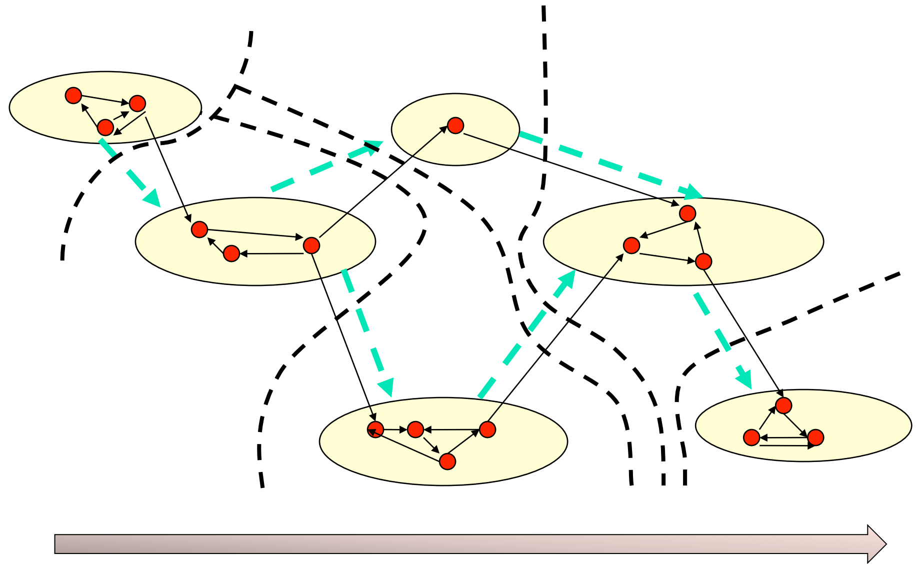
  - Imperatively: Depth first search in O(n+e)
- The AC has n strongly connected components

Acyclic Condensation

- The SCC of a graph form „abstract super nodes"
- That dag of super nodes is called **acyclic condensation (AC)**



- Many algorithms need acyclic graphs, in particular attribute evaluation algorithms
  - The data flow flows along the partial order of the nodes
  - For cyclic graphs, form an AC
- Propagate attributes along the partial order of the AC (*wavefront algorithm*)
  - Within an SCC compute until nothing changes anymore (fixpoint)
  - Then advance
  - No backtracking to earlier SCCs
- Evaluation orders are the topsorts of the AC



- SCCs can be made on every graph
  - Always a good structuring means for every kind of diagram in design
  - SCCs form "centers"
  - Afterwards, the AC can always be topsorted, i.e., evaluated in a total order that respects the dependencies
- Useful for structuring large
  - Data diagrams: Class diagrams, package diagrams, object diagrams
  - Behavioral diagrams: statecharts, data-flow diagrams, Petri nets, and UDUGs, call graphs
  - Coalesce loops into subdiagrams
- Wavefronts can be used for attribute calculations on graphs
  - Analyzing statistics on graphs
  - "reduce" problems: reducing all attributes of a specific kind over all nodes and edges of the graph
  - Flow problems: calculating costs of paths

- Computing definition-use graphs
  - Many diagrams allow to *define* a thing (e.g., a class) and to *use* it
  - Often, you want to see the graph of definitions and uses (the *definition-use* graph)
  - Definition-use graphs are important for refactoring, restructuring of software
    - Whenever a definition is edited, all uses must be adapted
    - A definition use graph refactoring tool automatically updates all uses
- Computing Software Metrics
  - A ***metric*** is a quantitative measure for code or models
  - Metrics are computed as attributes to source code entities, usually in a wavefront
  - Examples:
    - Number of instruction nodes in program graphs (instead of Lines-of-code)
    - Call graph depth (how deep is the call graph?)
    - Depth of inheritance dag (too deep is horrible)

Has the graph a skeleton tree structure? [ASU86]

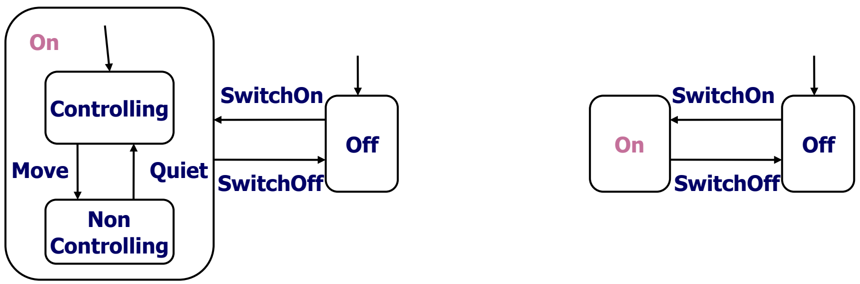
(Finding a hierarchy in a graph-based model)

16.3 REDUCIBILITY

Why Is a UML Statechart Simple to Understand?

- It is not a plain automaton
- But hierarchically organized
  - Certain states *abstract* substatecharts

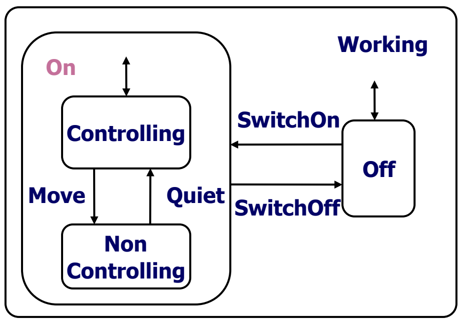
Auto Pilot

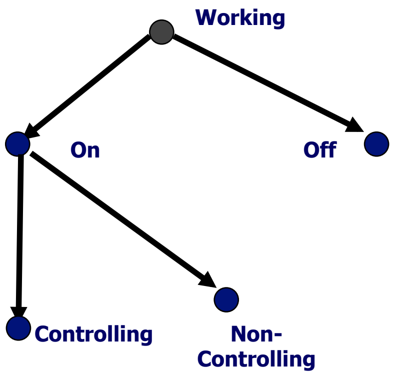


... it is a Reducible Graph

- But hierarchically organized

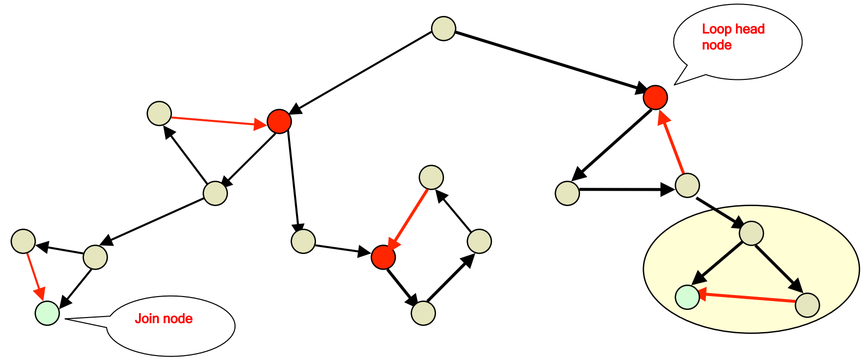
Auto Pilot





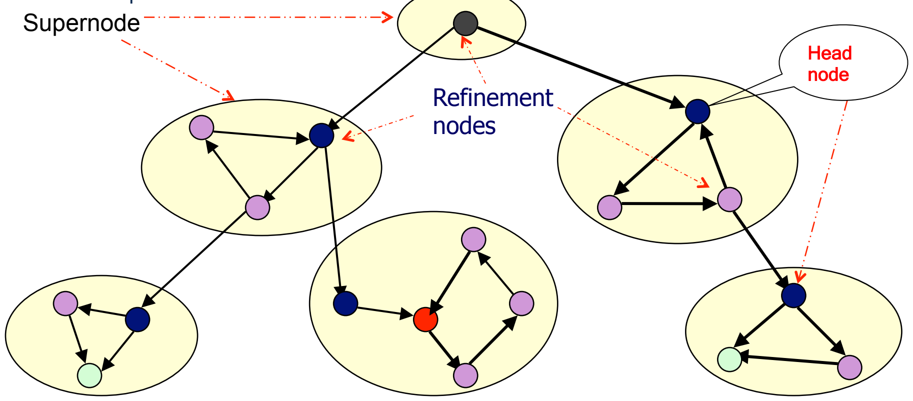
A Reducible Graph

- A reducible graph has special areas with subdags and cycles, *supernodes*
- In a reducible graph, there is a spanning tree with *primary edges*:
  - Each diamond has a secondary edge, ending in a *join node*
  - Each cycle has one backedge to a *loop head node*
- Attention: this is not an acyclic condensation!



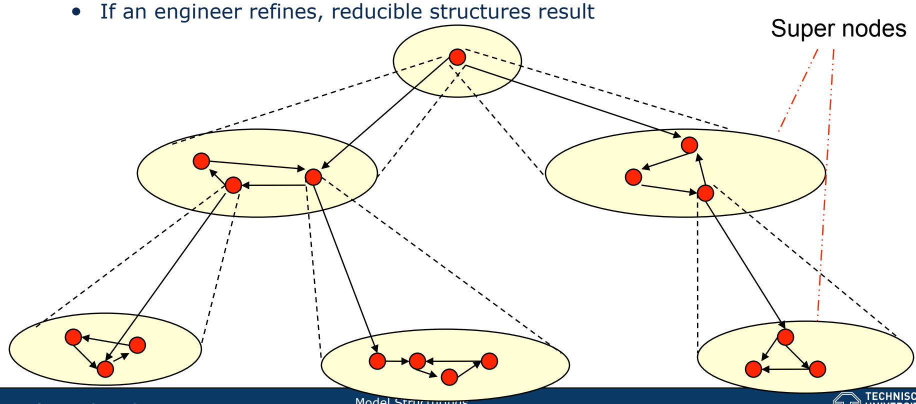
A Reducible Graph

- Every supernode has a *head* that *represents* or *abstracts* it
  - All ingoing edges into the super node end in the head
  - Loop head nodes can be head nodes; join nodes not
  - The head node of a supernode is refined from a *refinement node* in another supernode



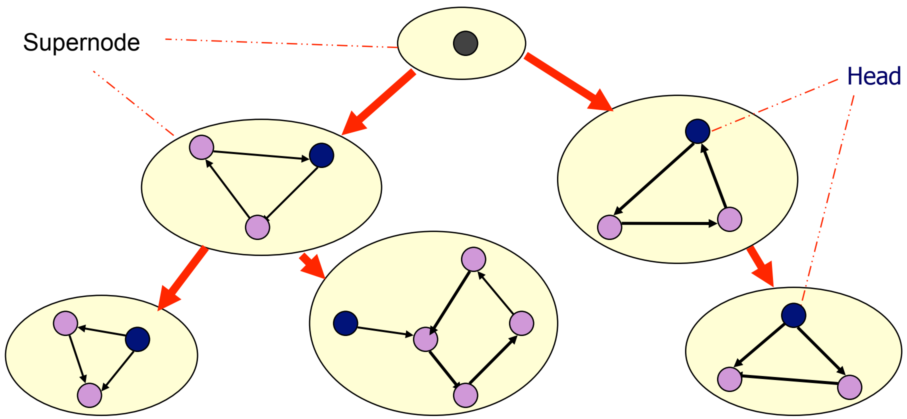
Reducible Graphs

- Reducible graphs have a hierarchical structure, expressed by their **skeleton tree of super nodes** with head nodes
  - Supernodes can hide subgraphs
  - Attention: SCC have a DAG structure (different!)
- Reducible graphs may stem from the *refinement operation applied to refinement nodes*
  - If an engineer refines, reducible structures result



A Reducible Graph

- A **skeleton tree (skeleton hierarchy)** between the supernodes results
- Graph is structured and much simpler to comprehend



Reducible Graphs in Software Engineering

- Submodels can be *abstracted* into single nodes
- Whole model can be abstracted into one node
- Skeleton tree structures the model
- Reducibility law:

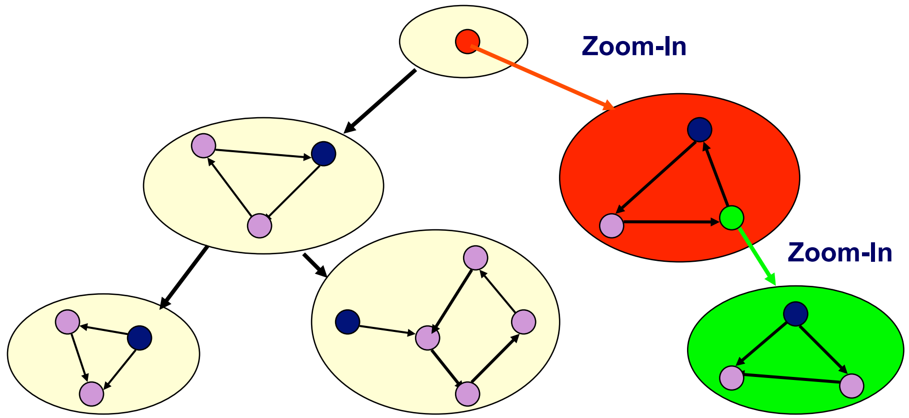
> A model *should* use reducible graphs
> to be comprehensilbe
> and to enable efficient algorithms

- Otherwise large models cannot be understood

> **Principle of structured modeling and structured programming:**
> The refinement operation is very helpful because
> it results in reducible graphs and models

The Fractal-Like Behavior of Reducible Graphs

- A reducible graph can be zoomed-in and zoomed-out, like a fractal
- Refinement nodes can be zoomed in
- Zooming-out means *abstraction*
- Zooming-in means *detailing*

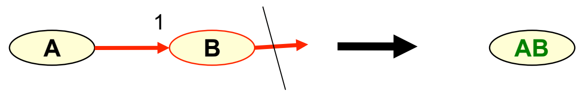


Computing Reducibility with T1-T2 Graph Rewrite System

- A *reducible* digraph is a digraph, that can be reduced to one node by the following graph rewrite rules [Tarjan74]
- Specification with a subtractive GRS (SGRS):

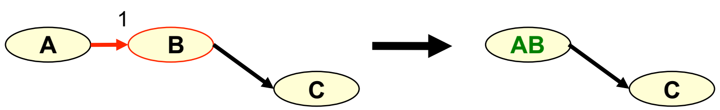
Reducibility-T1: Remove reflective edges

Reducibility-T2a: Merge successors with no fan-out and fan-in 1 (collapse rule a)



Computing Reducibility with T1-T2 Graph Rewrite System

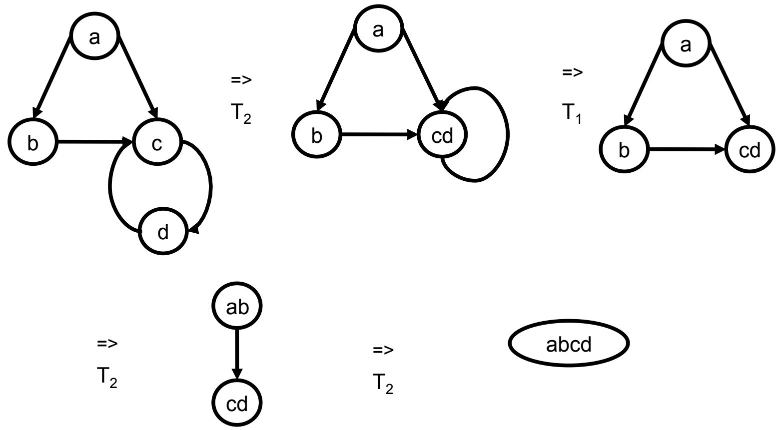
Reducibility-T2b: Merge successors with fan-in of 1 and fanout (collapse rule b)



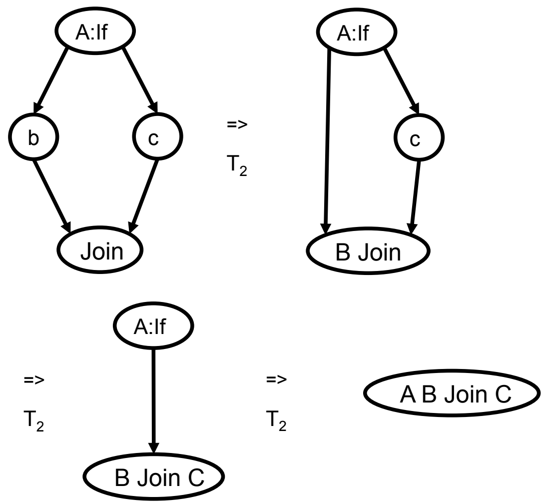
- Side condition of Reducibility-T_2: If there is a node *B*, that has a unique predecessor, *A*, then *m* may consume *n* by deleteing *B* and making all successors C of *B* (including *A*, possibly) be successors of A.

Example: T_1 – T_2 Reduction

- On every level, in the super nodes there may be cycles
  - T2 shortens these cycles
  - T1 reduces reflective cycles to super nodes
- Example: Reduction of a finite state automaton



- Reduction of an IF structure



Reducible Graphs

- All recursion techniques on trees can be taken over to the skeleton trees of the reducible graphs
- For reducible graphs, usually recursion schemas can be applied
  - Branch-and-bound
  - Depth-first search
  - Dynamic programming
- Applications
  - Organisation diagrams: if a organization diagram is not reducible, something is wrong with the organization
    - This is the problem of matrix organizations in contrast to hierarchical organizations
  - How to Diff a Specification?
    - Text: well-known algorithms (such as in RCS)
    - XML trees: recursive comparison (with link check)
    - Dags: layer-wise comparison
    - Graphs: ??? For general graphs, diffing is NP-complete (graph isomorphism problem)

Application: Simple Diffing in Reducible Graphs

- Given a difference operator on two nodes in a graph, there is a generic linear diff algorithm for a reducible graph:
  - Walk depth-first over both skeleton trees
  - Form the left-to-right spanning tree of an SCC and compare it to the current SCC in the other graph
- Exercises: effort?
  - how to diff two UML class diagrams?
  - how to diff two UML statecharts?
  - how to diff two colored Petri Nets?
  - how to diff two Modula programs?
  - how to diff two C programs?

Applications of Reducibility in Software Engineering

- ***Structured programming*** produces reducible control flow graphs (Modula and Ada, but not C)
  - Dijkstra's concern was reducibility
  - Decision tables (Entscheidungstabellen) sind hierarchisch
  - *Structured Analysis (SA)* is a reducible design method
  - *Colored Petri Nets* can be made reducible
  - UML
- CBSE Course:
  - *Component-connector diagrams* in architecture languages are reducible
  - Many component models (e.g., Enterprise Java Beans, EJB)
- *Architectural skeleton programming (higher order functional programming)*
  - Functional skeletons map, fold, reduce, bananas

Example: UML Restructuring

- Structure UML Class Diagrams
- Choose an arbitrary UML class diagram
- Calculate reducibility
  - If the specification is reducible, it can be collapsed into one class
  - Reducibility structure gives a simple package structure
- Test dag feature
  - If the diagram is a dag, it can be layered
- TopSort the diagram
  - A topsort gives a linear order of all classes
- UML Packages are not reducible per se
  - Large package systems can be quite overloaded
  - Layering is important (e.g., 3-tier architecture)
  - Reducible packages can be enforced by programming discipline. Then, packages can better be reused in different reuse contexts
- UML statecharts are reducible
- UML component, statecharts and sequence diagrams are reducible

Restructuring an arbitrary graph to be reducible

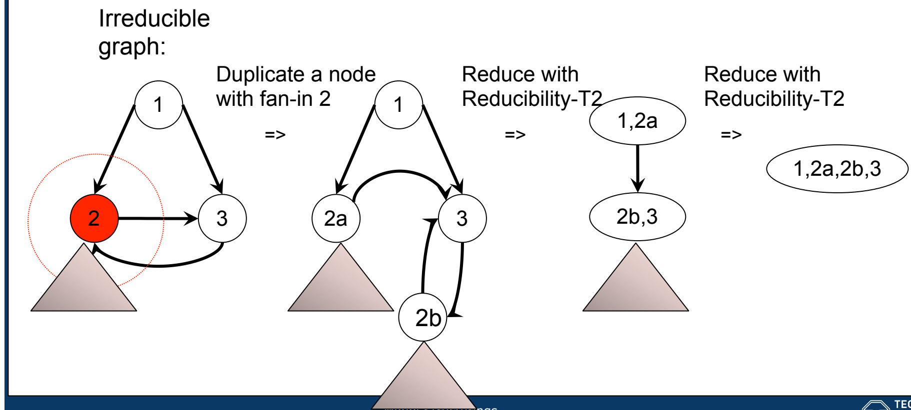
16.3.2 MAKING GRAPHS REDUCIBLE

Graphs Can Be Made Reducible

- By duplicating shared parts of the graph that destroy reducibility structure
  - Builds a skeleton tree
- The process is called *node splitting:*
  - If the reducability analysis yields a limit graph that is other than a single node, we can proceed by splitting one or more nodes
  - If a node *n* has *k* predecessors, we may replace *n* by *k* nodes.
  - The *i*th predecessor of *n* becomes the predecessor of n_i only, while all successors of *n* become successors of all the n_i's.

Example: Node Splitting

- If a loop is irreducible, node has two ancestors. For instance, a join node may also be a loop head node
- Remedy
  - Separate the loop from the join node
  - Duplicate the irreducible node in an irreducible loop (even with subtrees)
  - Most often, the join and loop head node can be taken



16.4 SUMMARY OF STRUCTURINGS

Structurings Producing Lists and Graphs

- **More Structurings Producing Lists**
- **Layering**
  - Overlaying a list of layers onto a dag
  - "same generation problem"
  - Standard Datalog, DL, EARS problem

- **More Structurings Producing Trees**
- **Dominance Analysis**
  - Overlays a dominator tree to a graph
  - A node dominates another if all paths go through it
  - Applications: analysis of complex specifications
- **Planarity**
  - Finds a skeleton tree for planar drawing
  - A graph is planar, if it can be drawn without crossings of edges
  - Computation with a reduction GRS, i.e., planarity is a different form of reducibility
  - Application: graph drawing
- **Graph parsing with context-free graph grammars**
  - Overlaying a derivation tree
  - Rules are context-free

More Structurings Producing Dags

- Stratification
  - Layers of graphs with two relations
  - Normal (cheap) and dangerous (expensive) relation
  - The dangerous relation must be acyclic
  - And is layered then
  - Applications: negation in Datalog, Prolog, and GRS
- Concept Analysis [Wille/Ganter]
  - Structures bipartite graphs by overlaying a lattice (a dag)
  - Finds commonalities and differences automatically
  - Eases understanding of concepts

Comparison of Structurings

	List	Tree	Dag	Concept	Purpose
TopSort	x			Order	Implementation of process diagrams
Layering	x			Order	Layers
Reducibility		x		Hierarchy	Structure
Dominance		x		Importance of nodes	Visit frequency
Planarity		x		Hierarchy	Drawing
Graph parsing		x		Hierarchy	Structure
Strongly conn. components			x	Forward flow Wavefronts	Structure
Stratification			x	Layering	Structure
Concept analysis			x	Commonalities	Comparison

Simple Models in Software Engineering

- Models and specifications, problems and systems are easier to understand if they are
  - Sequential
  - Hierarchical
  - Acyclic
  - Structured (reducible)
- And this hold for every kind of model and specification in Software Engineering
  - Structurings can be applied to make them simpler
  - Structurings are applied in all phases of software development: requirements, design, reengineering, and maintenance
  - Forward engineering: define a model and test it on structure
  - Reverse engineering: apply the structuring algorithms

Other Software Engineering Applications

- Structured Programming (reducible control flow graphs), invented from Dijkstra and Wirth in the 60s
- Description of software architectures (LeMetayer, 1995)
- Description of refactorings (Fowler, 1999)
- Description of aspect-oriented programming (Aßmann/Ludwig 1999)
- Virus detection in self-modifying viruses

The End: What Have We Learned

- Understand Simon’s Law of Complexity and how to apply it to graph-based models
- Techniques for treating large requirements and design models
- Concepts for *simple* software models
- You won't find that in SE books
- .... but it is essential for good modelling in companies