TECHNISCHE UNIVERSITÄT DRESDEN

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

14. 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 11-0.2, 12.11.11

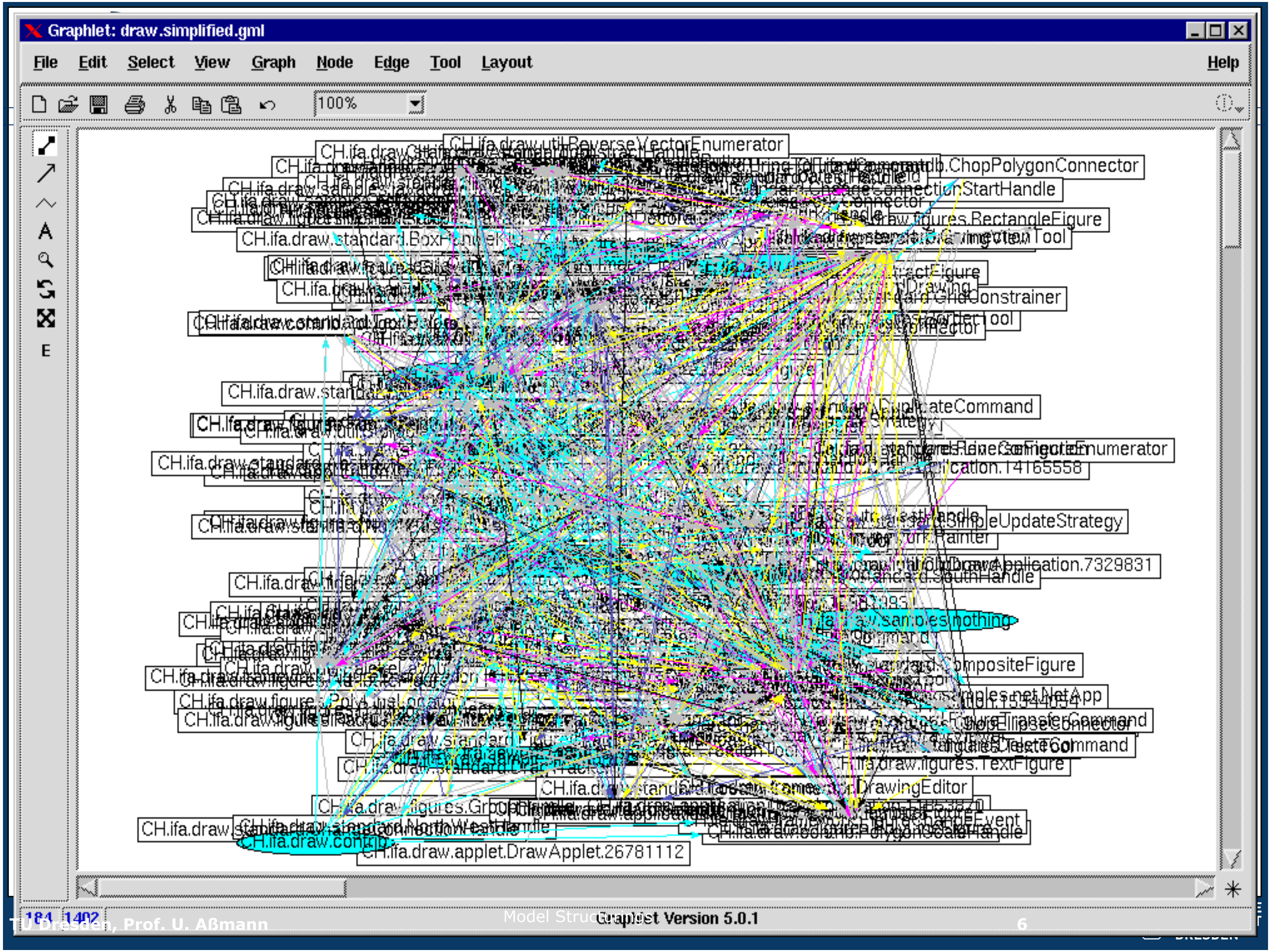
1. Graph Structurings with Graph Transformations
2. Additive and Subtractive GRS (extern)
3. Triple Graph Grammars
4. Graph Structurings
   1. Layering
   2. Strongly Connected Components
   3. Reducibility
5. Summary of Structurings

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/
- T. Mens. On the Use of Graph Transformations for Model Refactorings. In GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/
- www.fujaba.de www.moflon.org
- 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/TAGT1998.pdf

Further Reading

- Reducible graphs
- 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


Graphlet: draw.simplified.gml
File
Edit
Select
View
Graph
Node
Edge
Tool
Layout
Help
100%
CH.ifa.draw.util.ReverseVectorEnumerator
CH.ifa.draw.figures.RectangleFigure
CH.ifa.draw.standard.BoxHandleKit
CH.ifa.draw.applet.DrawApplet.26781112
CH.ifa.draw.figures.TextFigure
CH.ifa.draw.figures.PolyLineLocator
CH.ifa.draw.contrib
Graphlet Version 5.0.1
184
1402

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]

- 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

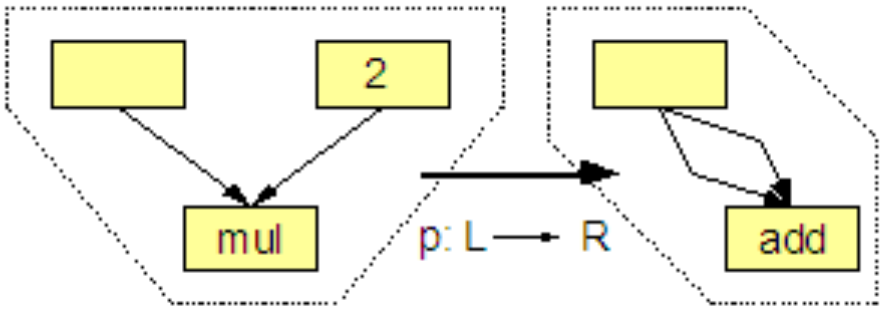
14.1 GRAPH TRANSFORMATIONS FOR GRAPH STRUCTURING

Idea: Structure the Software Systems With Graph Rewrite Systems

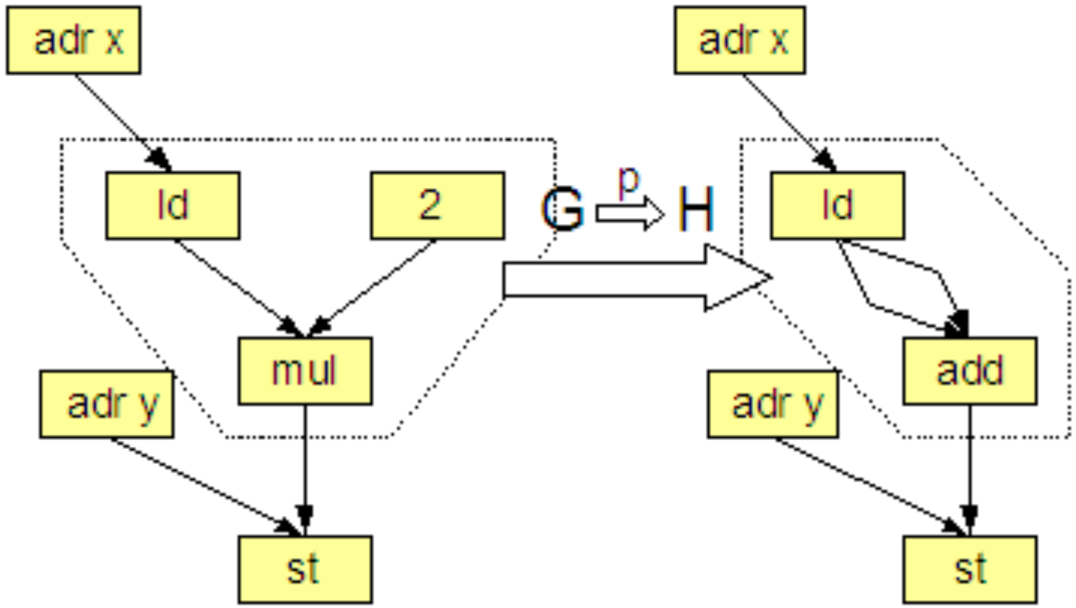
- Once, we do not only manipulate edges, but also nodes, we leave the field of Edge Addition Rewrite Systems
- We arrive at general Graph Rewrite Systems (GRS)
  - Transformation of complex structures to simple ones
  - Structure complex models and systems

- A *graph rewrite system* G = (S) consists of
  - A set of rewrite rules S
    - A rule r = (L,R) consists of 2 graphs L and R (left and right hand side)
    - Nodes of left and right hand side must be identified to each other
    - L = "Mustergraphen" ; R = Ersetzungsgraph"
  - An application algorithm A, that applies a rule to the manipulated graph
    - There are many of those application algorithms…
- A *graph rewrite problem* P = (G,Z) consists of
  - A graph rewrite system G
  - A start graph Z
  - One or several result graphs
  - A derivation under P consists of a sequence of applications of rules (direct derivations)
- GRS offer automatic graph rewriting
  - A GRS applies a set of Graph rewrite rules until nothing changes anymore (to the fixpoint, chaotic iteration)
  - Problem: Termination and Uniqueness of solution not guaranteed

Rule

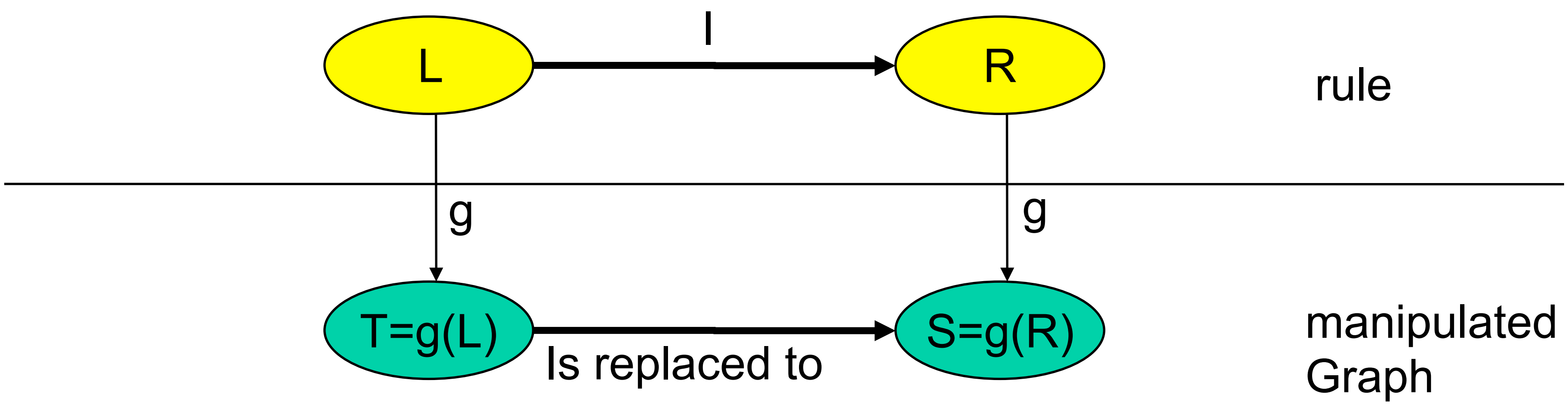


Redex in manipulated Graph G is rewritten to H



Application of a Graph Rewrite Rule

- **Match** the left hand side: Look for a subgraph T of the manipulated graph: look for a graph morphism g with g(L) = T
- Evaluate **side conditions**
- Evaluate right hand side
  - Delete all nodes and edges that are no longer mentioned in R
  - Allocate new nodes and edges from R, that do not occur in L
- **Embedding**: redirect certain edges from L to new nodes in R
  - Resulting in S, the mapping of g(R)

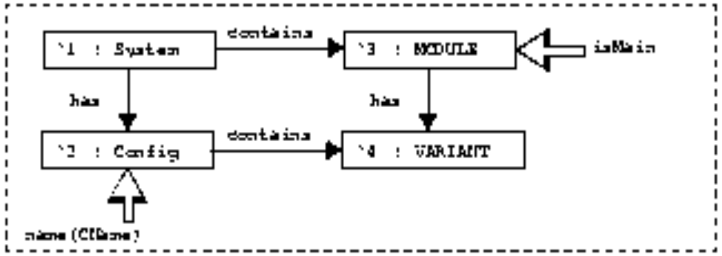


PROGRES, the GRS tool from the IPSEN Project

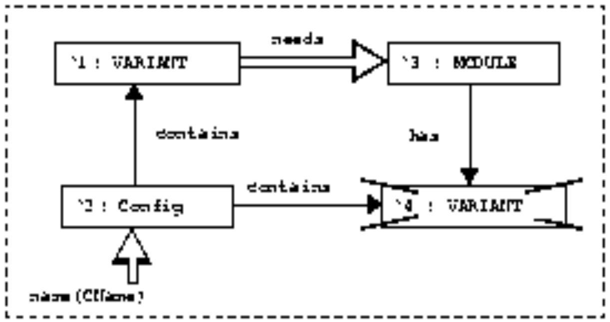
- PROGRES is a wonderful tool to model graph algorithms by graph rewriting
- Textual and graphical editing
- Code generation in several languages

- http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page_ref_id=213

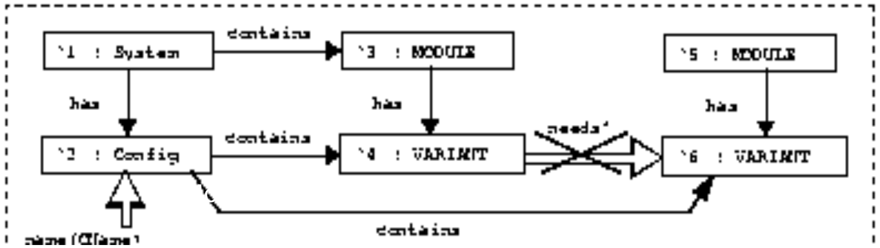
```
query ConsistentConfiguration( out CName : string ) =

   (* A configuration is consistent if:
   (* 1) it contains a variant of the system's main module, *)
   (* 2) it contains a variant for any module which is      *)
   (*    needed by another included variant, and            *)
   (* 3) it does not contain variants which are not needed  *)
   (*    by needed variants.                                *)

   use LocalName: string do
      ConfigurationWithMain( out LocalName )
    & not UnresolvedImportExists( LocalName )
    & not ConfigurationWithUselessVariant( LocalName )
    & CName := LocalName
   end

end;

test ConfigurationWithMain( out CName : string ) =
```




```
end;

test UnresolvedImportExists( CName : string ) =
```

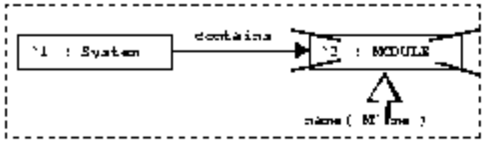



```
end;

test ConfigurationWithUselessVariant( CName : string ) =
```

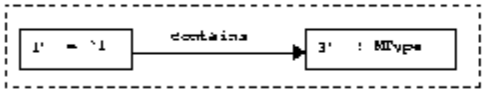


This example illustrates the possibilities of PROGRES to define *parametrized productions* which must be instantiated (in the sense of a procedure call) with actual attribute values and node types. In this way, a single production may abstract from a set of productions which differ only with respect to used attribute values and types of matched or created nodes. In almost all cases, node type parameters are not used for matching purposes, but provide concrete types for new nodes of the right-hand side.

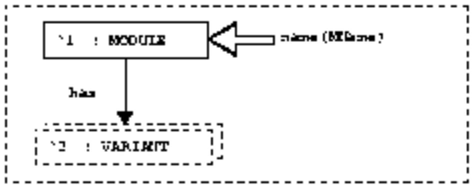
```
production CreateModule ( MName : string; InterfaceDescription : file;
                          MType : type in MODULE ) =
```



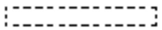

```
  ::=
```



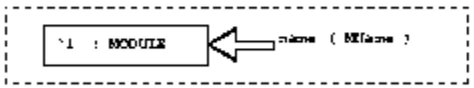

```
  transfer 2'.Name := MName;
           2'.File := InterfaceDescription;
end;

production DeleteModule ( MName : string ) =
```



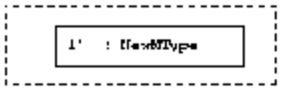

```
  ::=
```



```
end;

production ChangeModuleType ( MName : string; NewMType : type in MODULE ) =
```



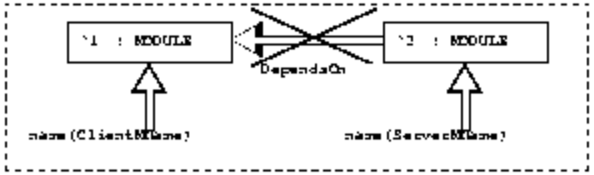

```
  ::=
```




```
  embedding redirect  -m_uses->, -has->, <-contains-,
                      <-m_uses-, <-v_uses- from '1 to 1';
  transfer 1'.Name := '1.Name;
           1'.File := '1.File;
end;
```

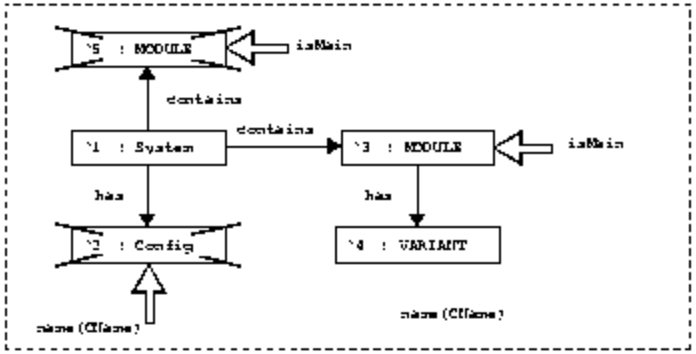
Fig. 12: Specification of basic graph transformations

```
production CreateMUse( ClientMName, ServerMName : string ) =
```

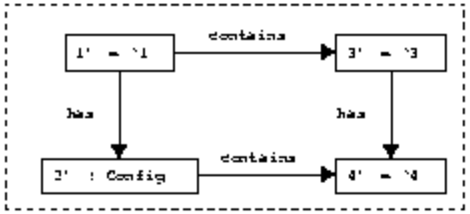



```
::=
```


```
end;
production InitConfig(    CName : string;
                          ReqProps : string [0:n];
                      out NewProps : string [0:n] ) =
```




```
::=
```




```
   condition '4.Props 'are_in' ReqProps;
   transfer  2'.Name := CName;
             2'.Props := ReqProps;
   return    NewProps := merge( ReqProps, '4.Props );
end;
```

Fig. 14: Specification of additionally needed complex productions

-24-

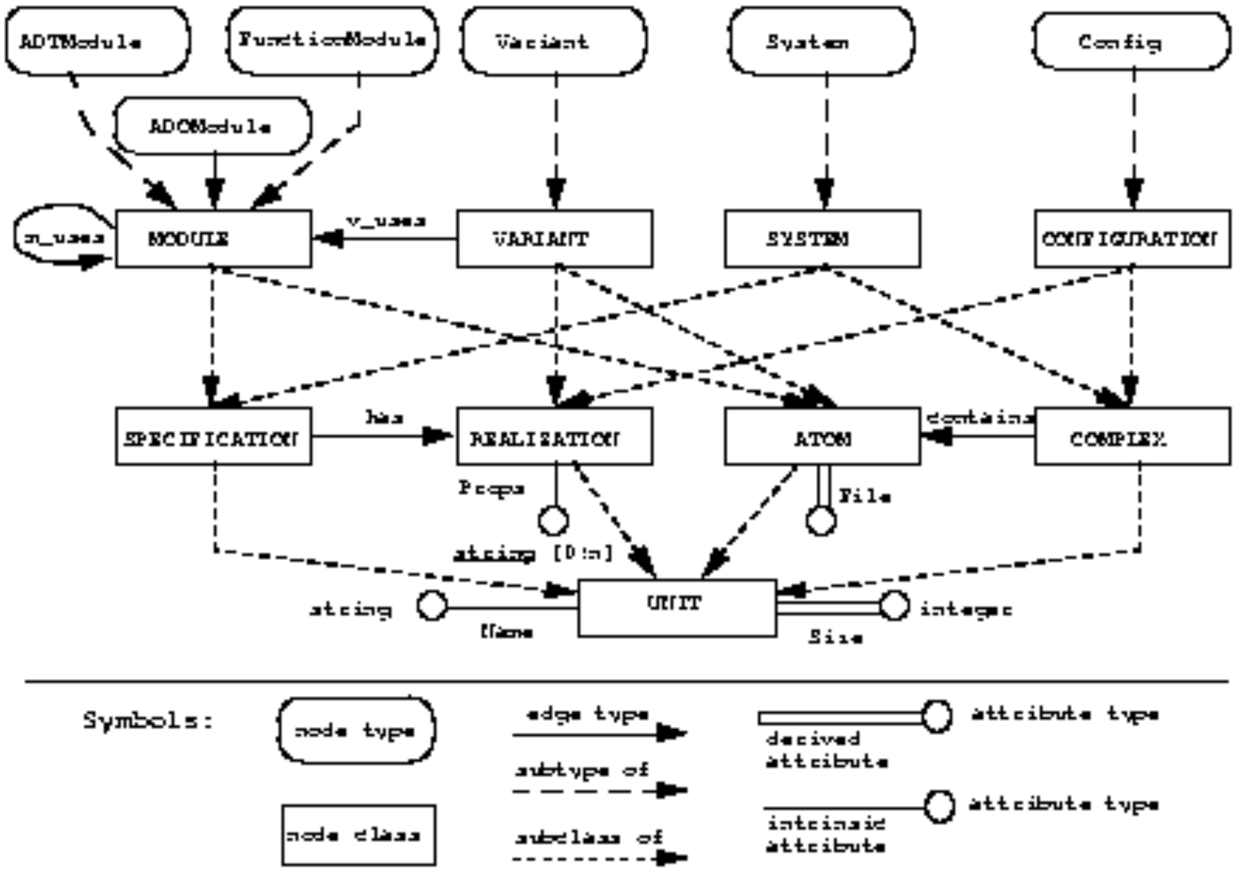


Fig. 5: The graph schema of MIL graphs (without derived relationships)

- *Boxes* with *round corners* represent node types which are connected to their uniquely defined classes by means of *dashed edges* representing "type is instance of class" relationships; the type ADTModule belongs for instance to the class MODULE.
- *Solid edges* between node classes represent edge type definitions; the edge type v_uses is for instance a relationship between VARIANT nodes and MODULE nodes and m_uses edges connect MODULE nodes with other MODULE nodes.
- *Circles* attached to node classes represent attributes with their names above or below

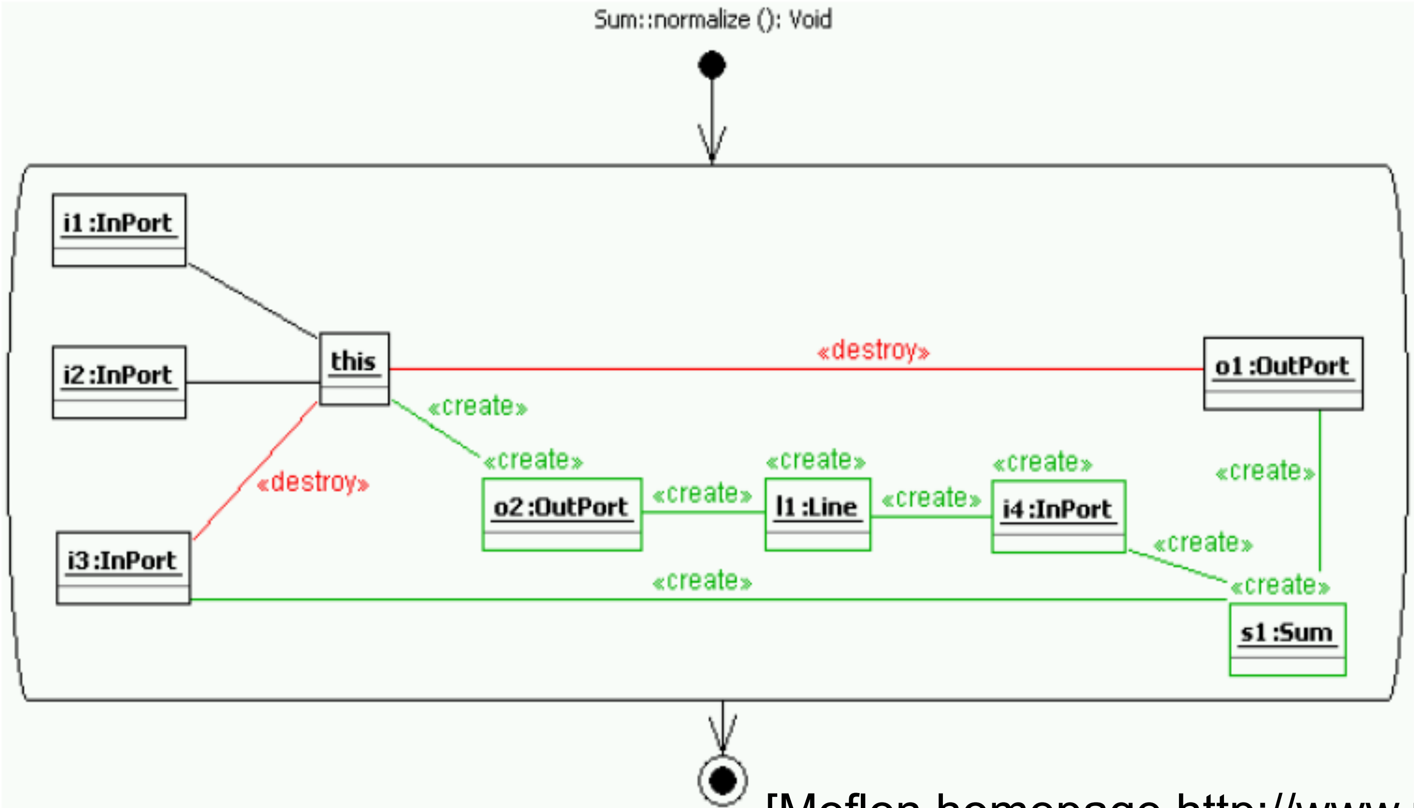
Different Kinds of Graph Transformation Systems

- Automatic Graph Rewriting
  - Iteration of rules until termination
- Programmed Graph Rewriting
  - The rules are applied of a control flow program. This program guarantees termination and selects one of several solutions
  - Examples: PROGRES from Aachen/München
  - Fujaba on UML class graphs, from Paderborn, Kassel www.fujaba.de
  - MOFLON from Darmstadt www.moflon.org
- Graph grammars
  - Special variant of automatic graph rewrite systems
  - Graph grammars contain in their rules and in their generated graphs special nodes, so called non-terminals
  - A result graph must not have non-terminals
  - In analogue to String grammars, derivations can be formed and derivation trees

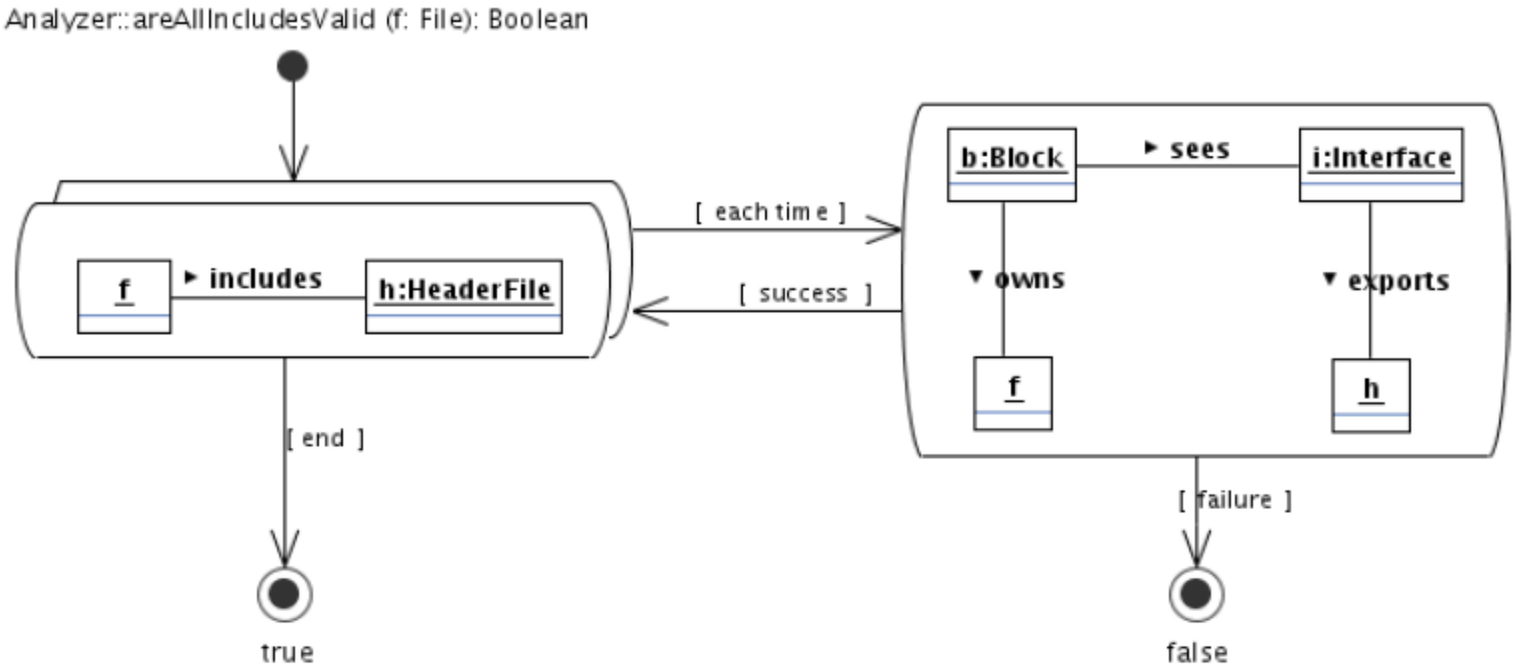
Different Kinds of Transformation Systems: TRS and DAGRS

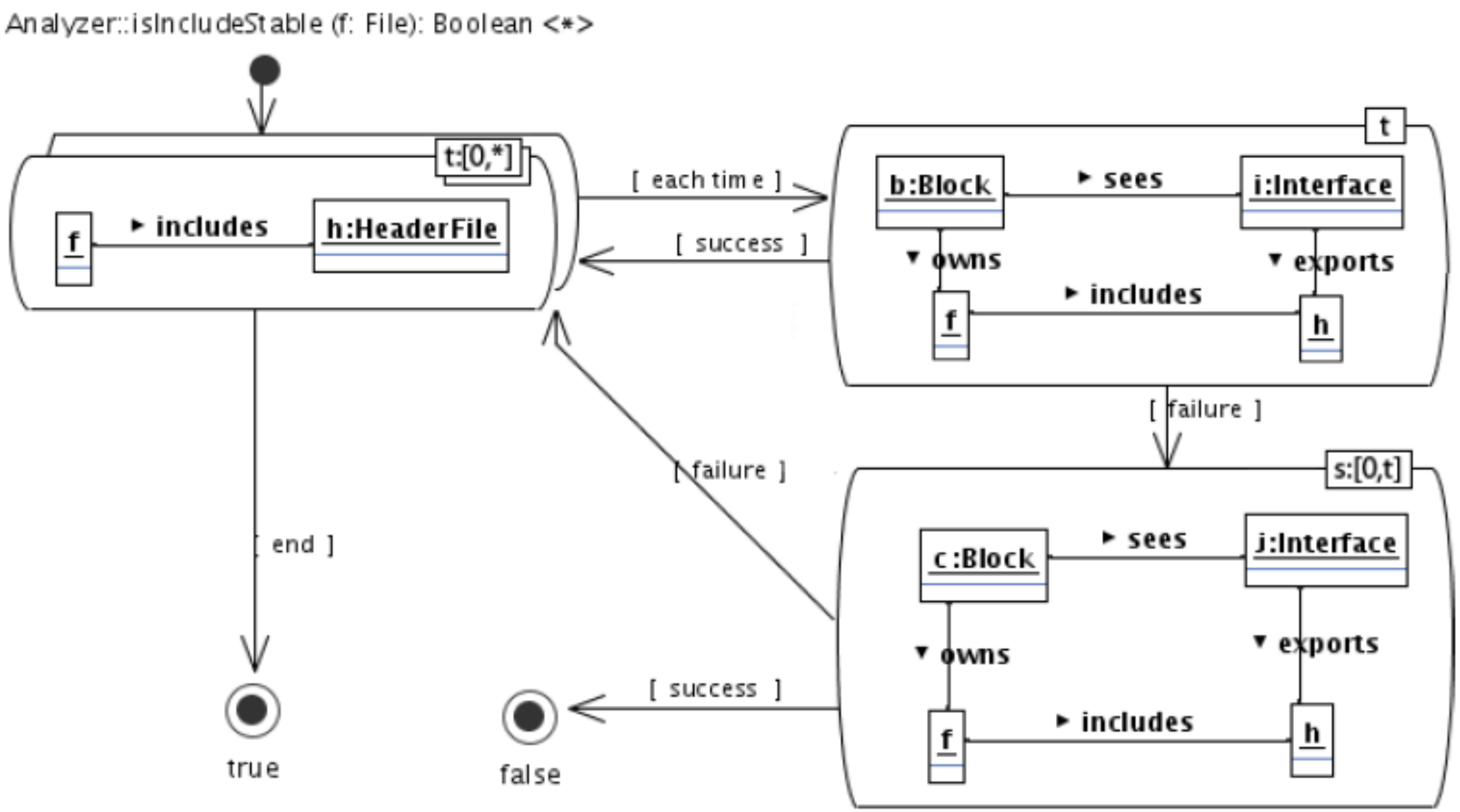
- Term rewriting replaces terms (ordered trees)
  - right and left hand sides are Terms
- Ground term rewrite systems, GTRS: only ground terms in left hand sides
  - A GTRS always works bottum-up on the leaves of a tree
  - For GTRS there are very fast, linear algorithms
- Variable term rewrite systeme, VTRS: terms with variables
  - Replacement everywhere in the tree
- Dag rewrite systems (DAGRS)
  - If a term contains a variable twice (non-linear), it specifies a dag
  - Dag rewrite systems containt dags in left and right hand sides (non-linear term rewriting)

- MOFLON and Fujaba embed graph rewrite rules into activity diagrams (aka storyboards)
  - A rule set executes as an atomic activity
  - Colors express actions

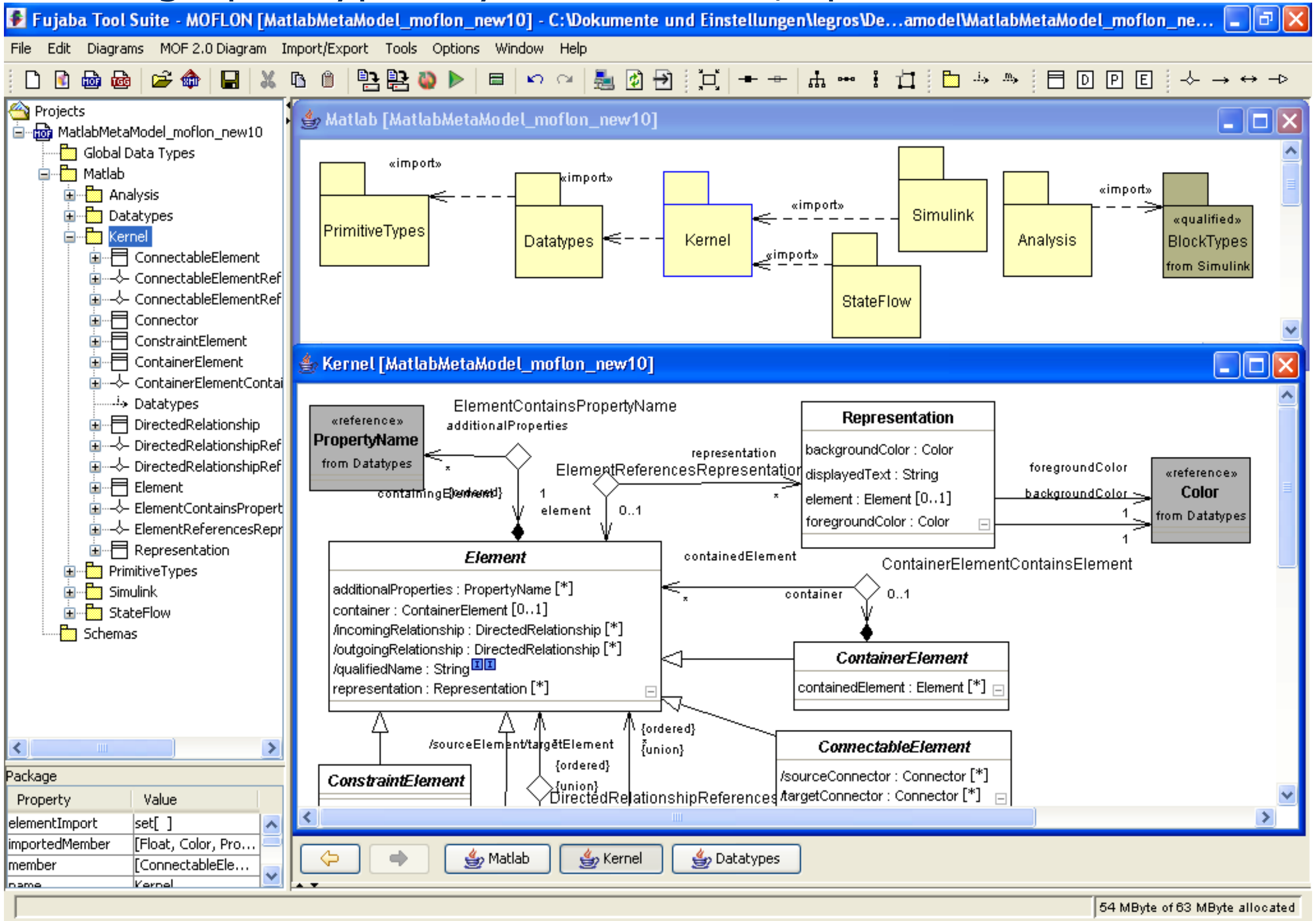


[Moflon homepage http://www.moflon.org]


Analyzer::areAllIncludesValid (f: File): Boolean
f
▸ includes
h:HeaderFile
[each time]
[success]
b:Block
▸ sees
i:Interface
▾ owns
▾ exports
f
h
[end]
[failure]
true
false


Analyzer::isIncludeStable (f: File): Boolean <*>
t:[0,*]
f
includes
h:HeaderFile
[each time]
[success]
t
b:Block
sees
i:Interface
owns
exports
includes
f
h
[failure]
[failure]
s:[0,t]
c:Block
sees
j:Interface
owns
exports
includes
f
h
[end]
[success]
true
false

- Works on graphs typed by metamodels, specified in MOF



Externer Foliensatz

14.2 KANTEN-ERSETZUNGSSYSTEME

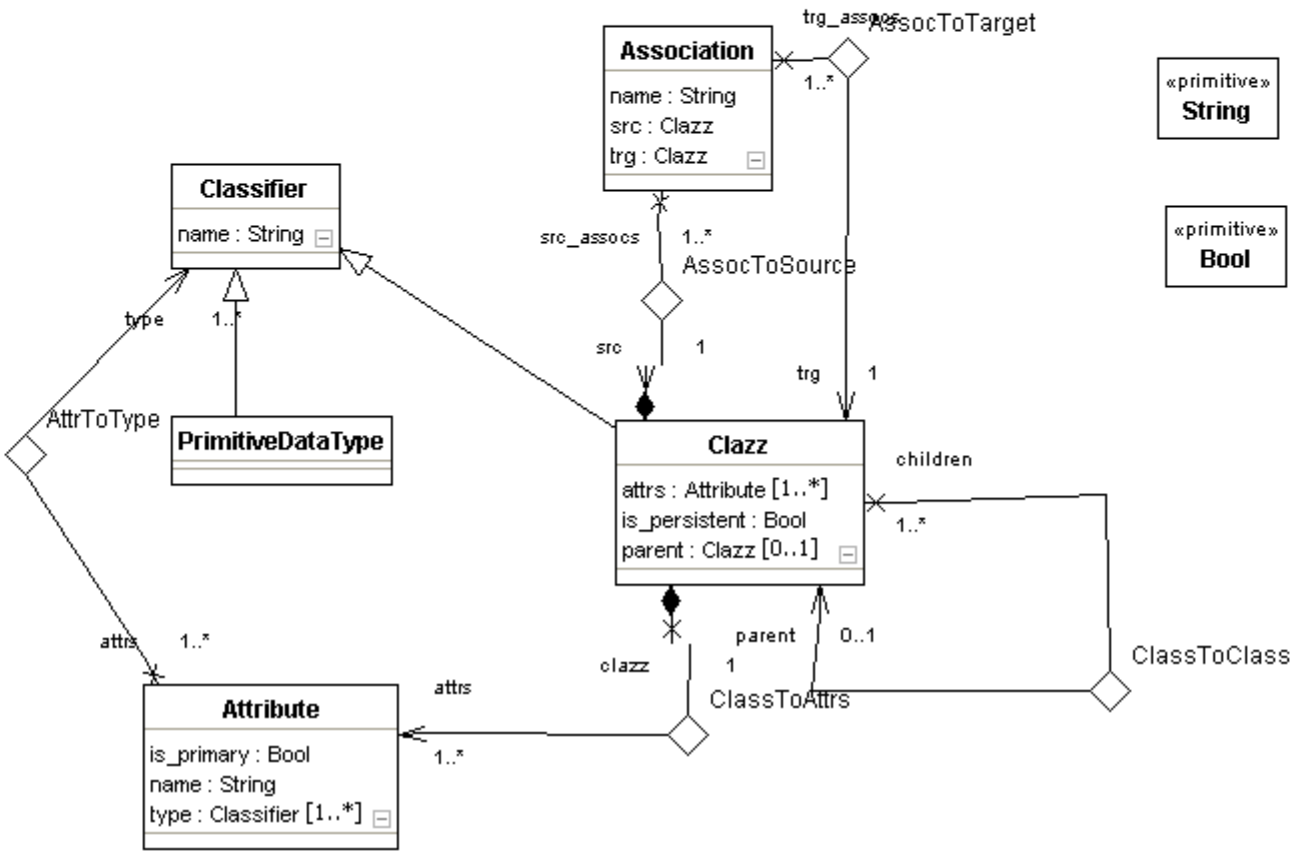
Mapping graphs to other graphs

Specification of mappings with mapping rules

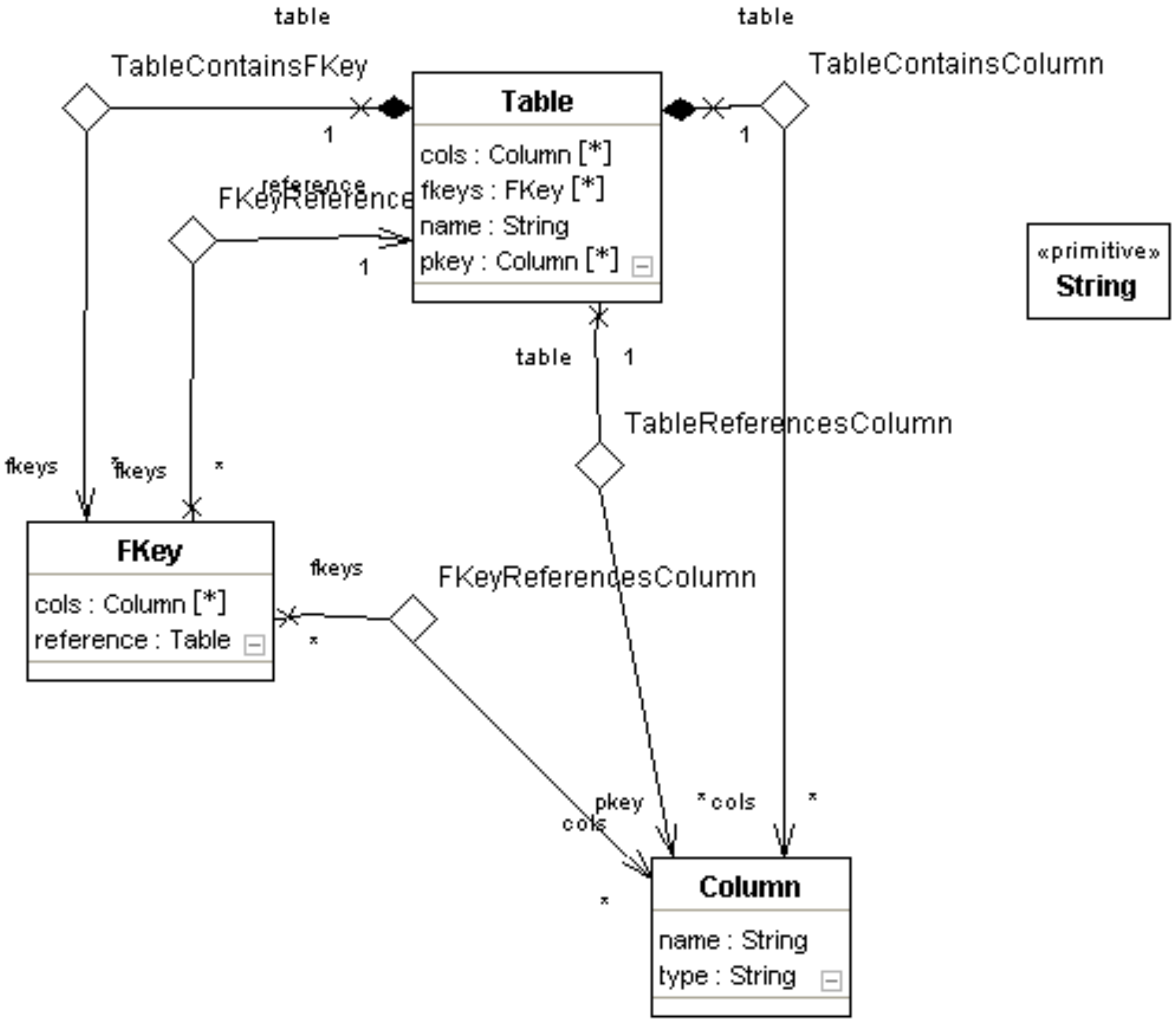
- Incremental transformation
- Traceability

14.3 TRIPLE GRAPH GRAMMARS

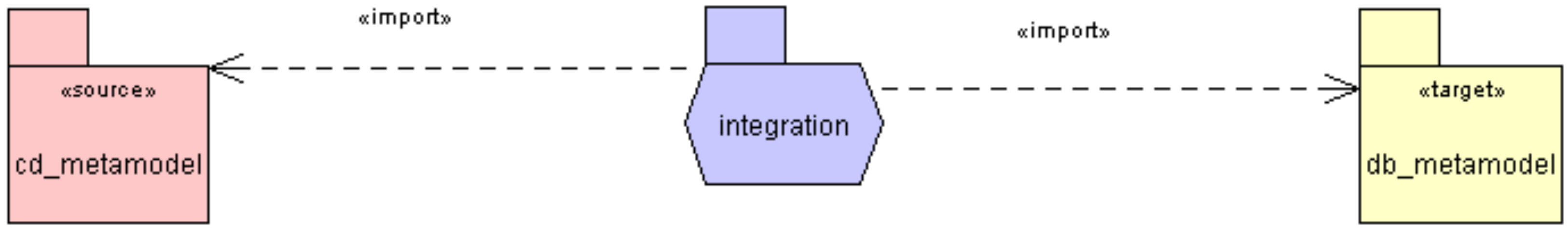
- Class diagram metamodel (CD)



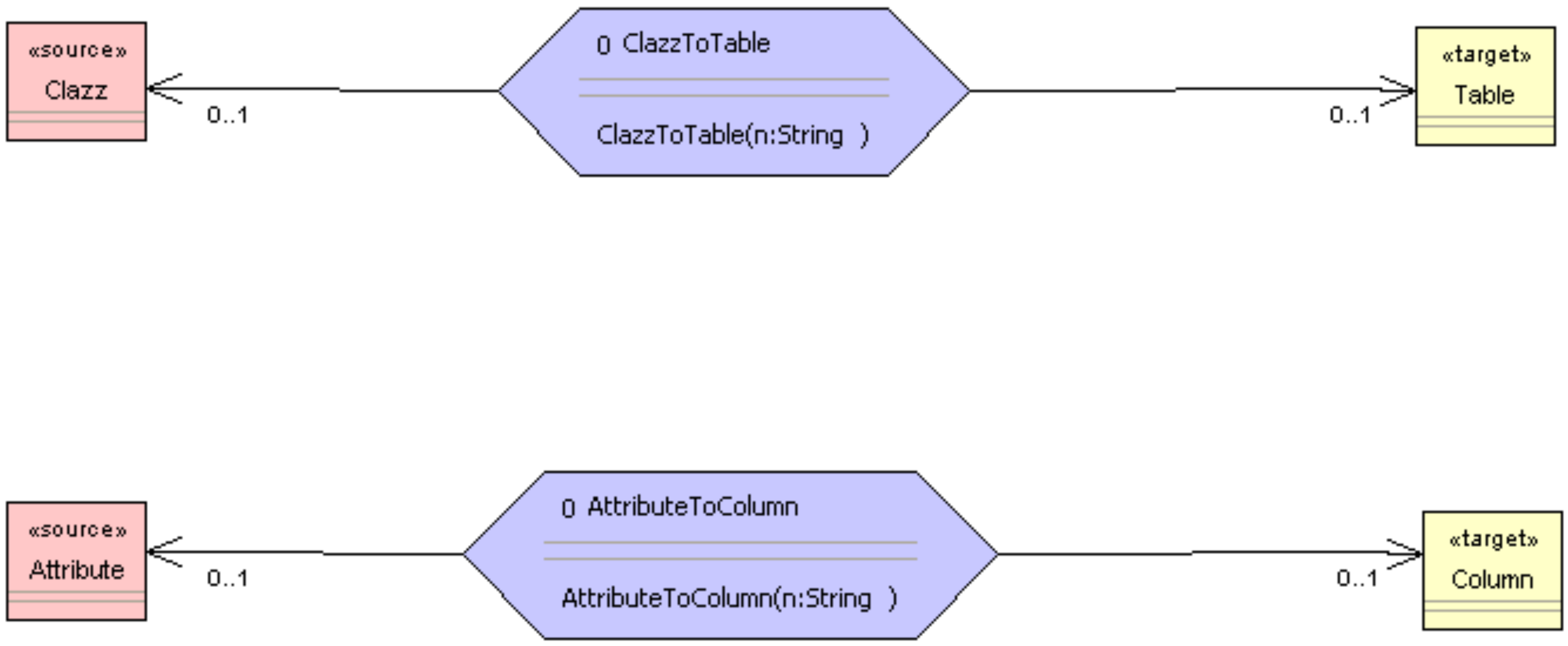
- Relational metamodel (db)

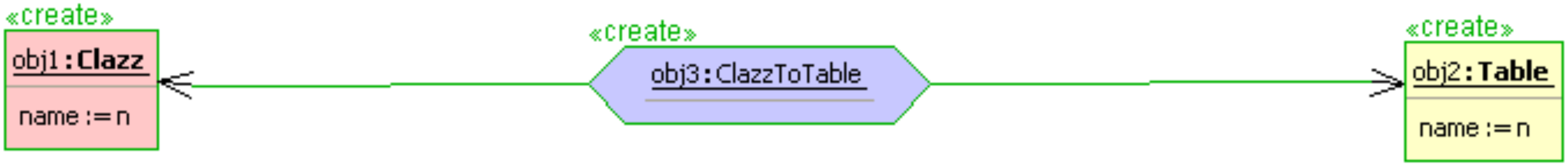


Triple Graph Grammars – Moflon Example



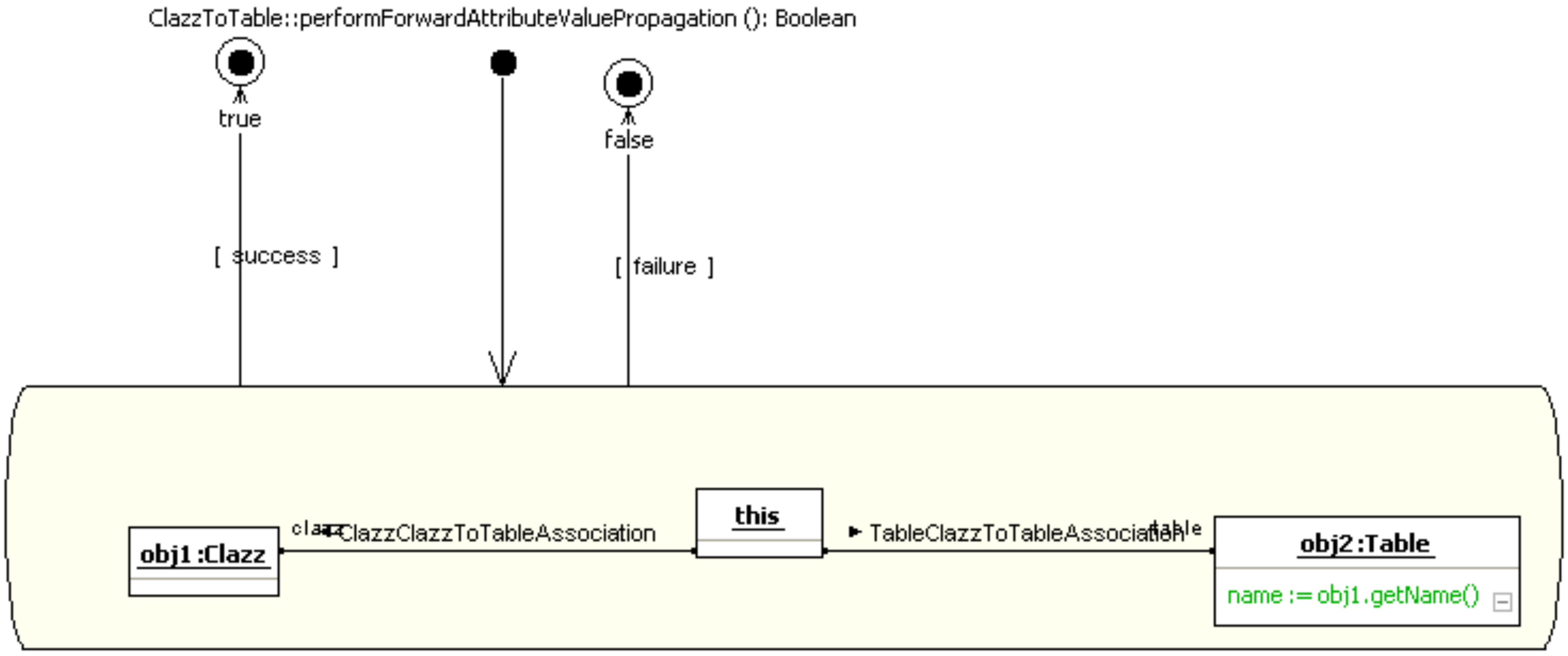
Triple Graph Grammars – Moflon Example



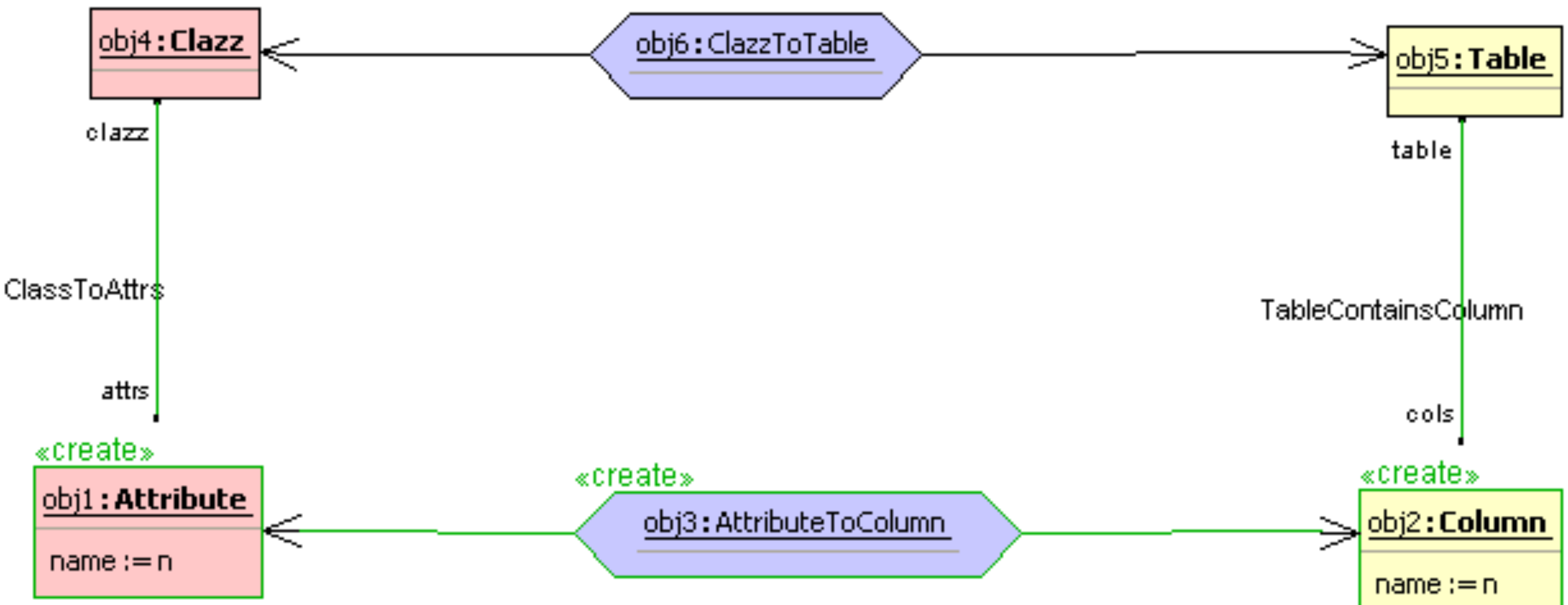


«create»
obj1 : Clazz
name := n
«create»
obj3 : ClazzToTable
«create»
obj2 : Table
name := n

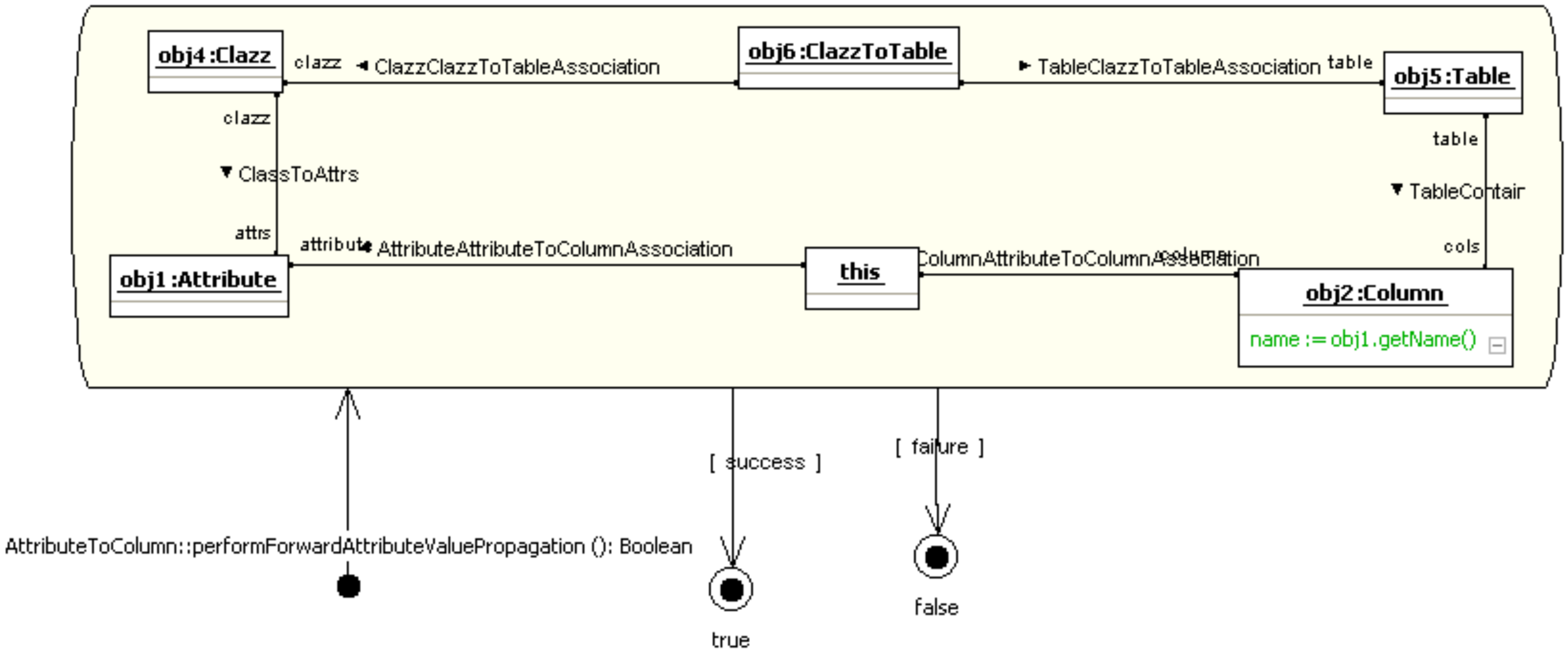
Triple Graph Grammars – Moflon Example



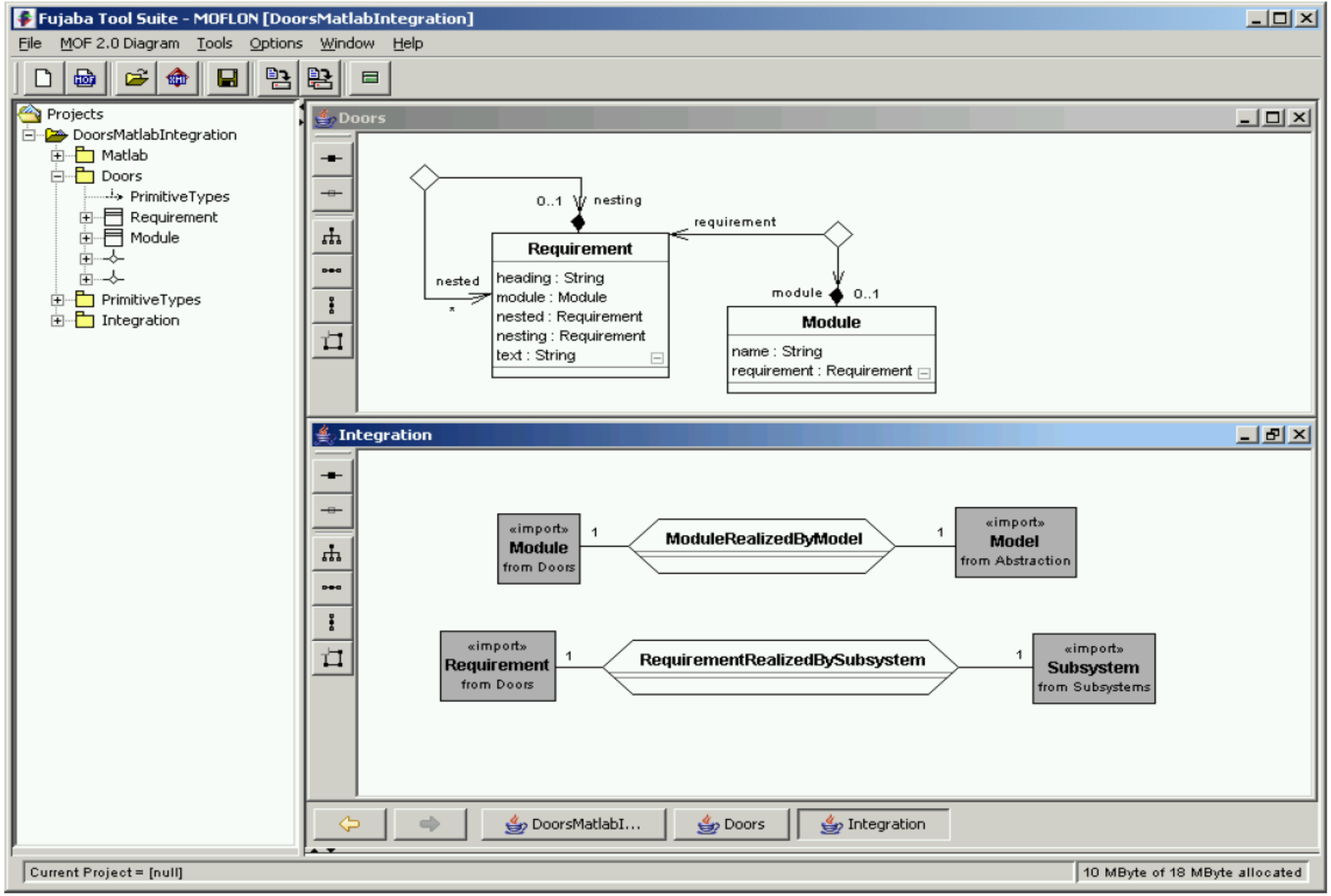
Triple Graph Grammars – Moflon Example



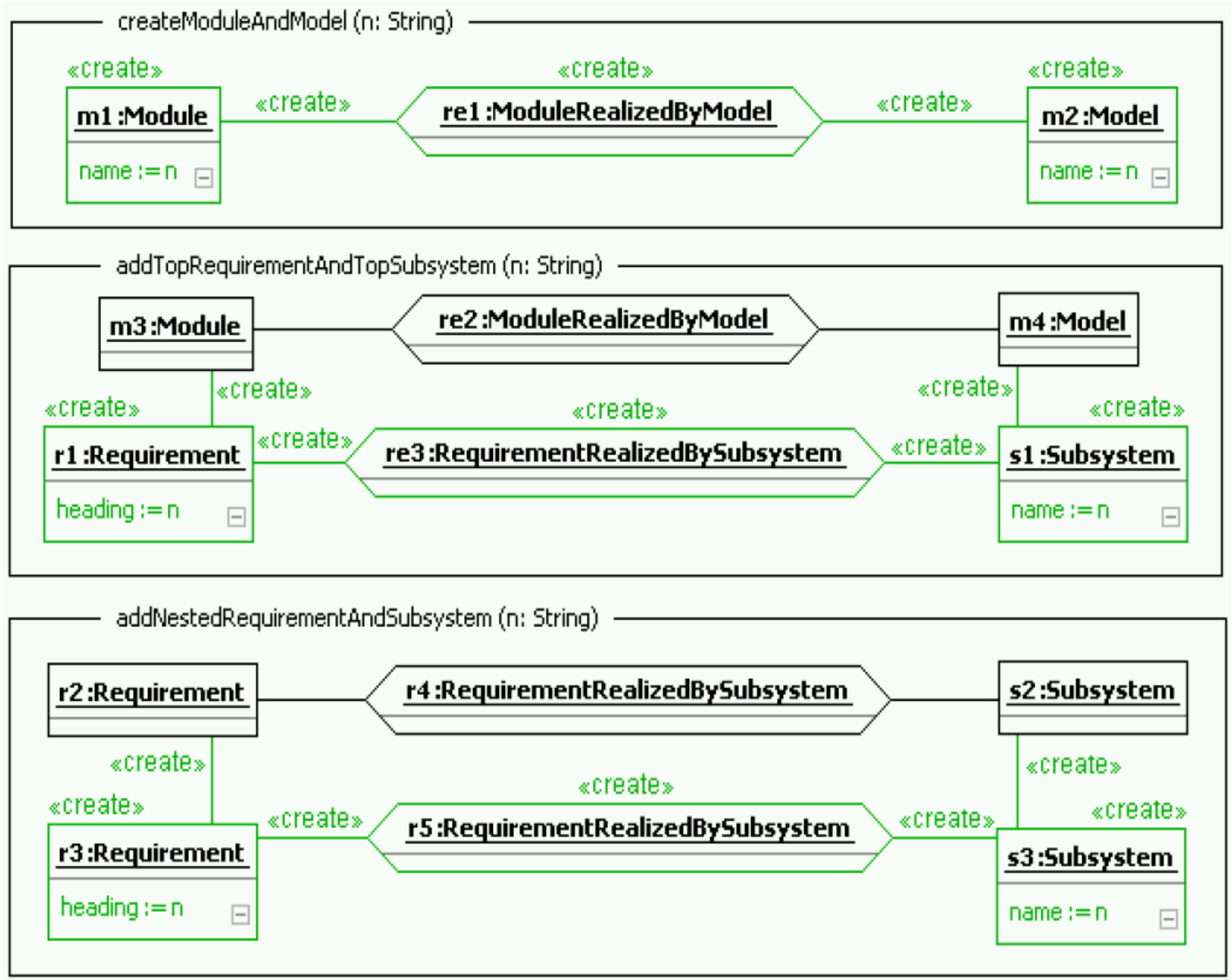
Triple Graph Grammars – Moflon Example



TGG Coupling Requirements Specification and Design

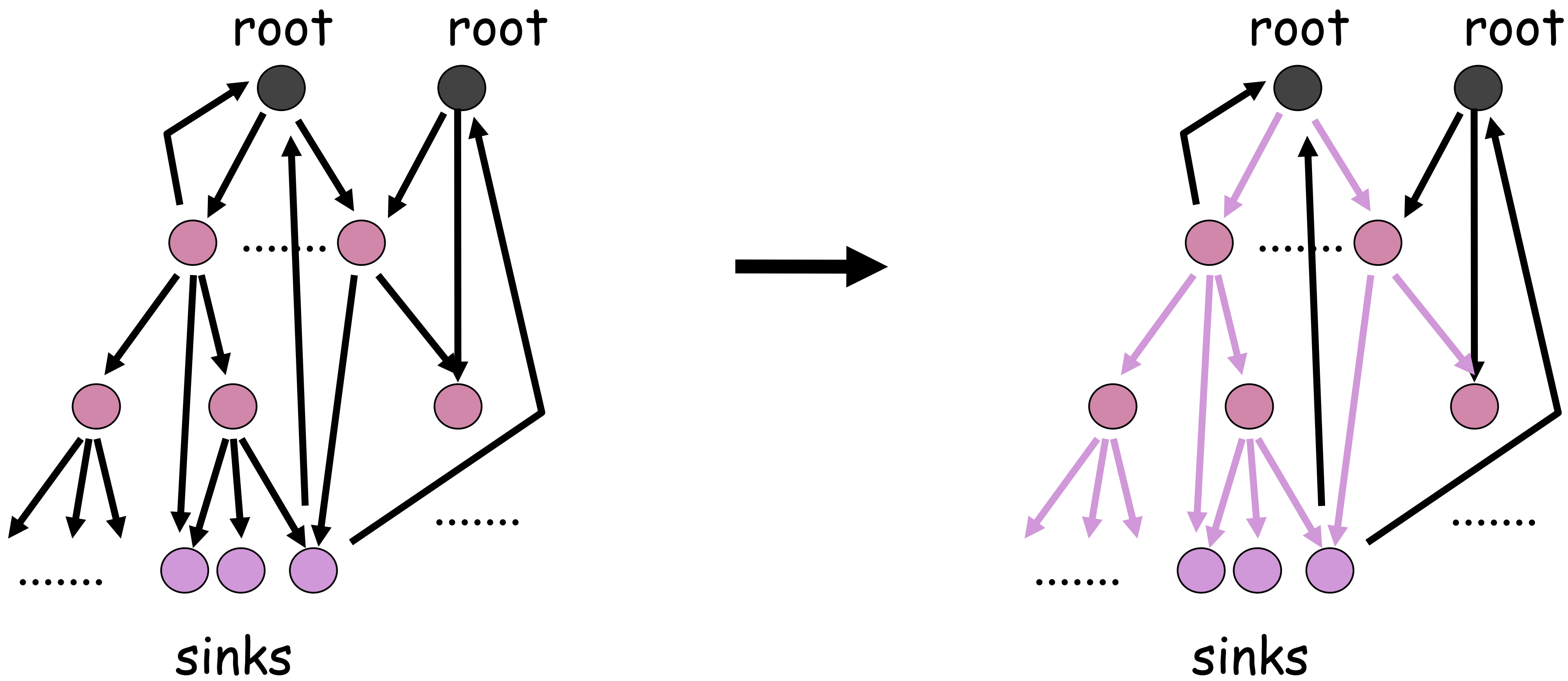


TGG Coupling Requirements Specification and Design



14.4 STRUCTURINGS OF GRAPHS

- Structurings *overlay* graphs with *skeleton lists, trees, and dags*

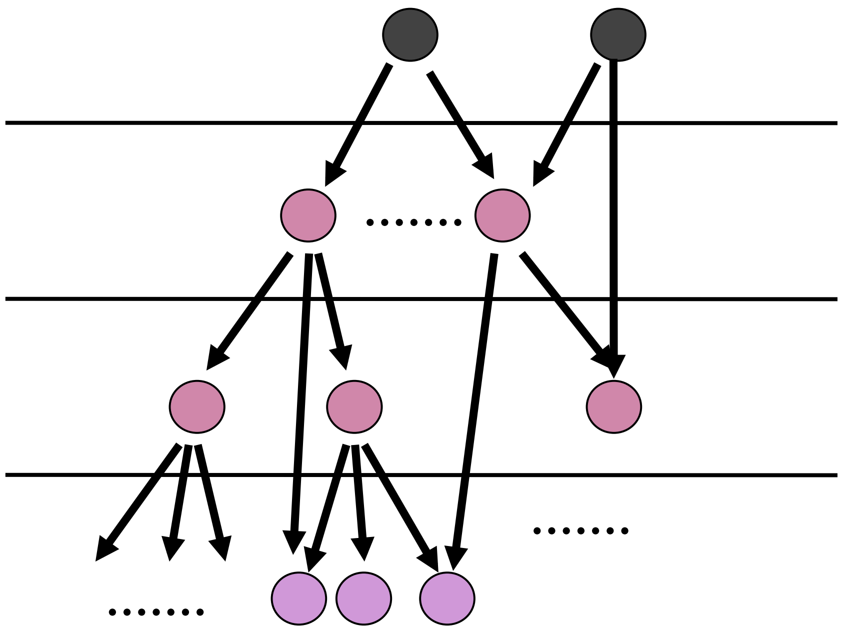


- Then, following the structure,
  - Sequential algorithms can be applied
  - Recursive algorithm schemas can be applied
  - Wavefronts can be applied
- Structures are nice for thinking and abstraction
  - In particular in analysis and design
- Structuring need
  - graph reachability analysis
  - graph transformation

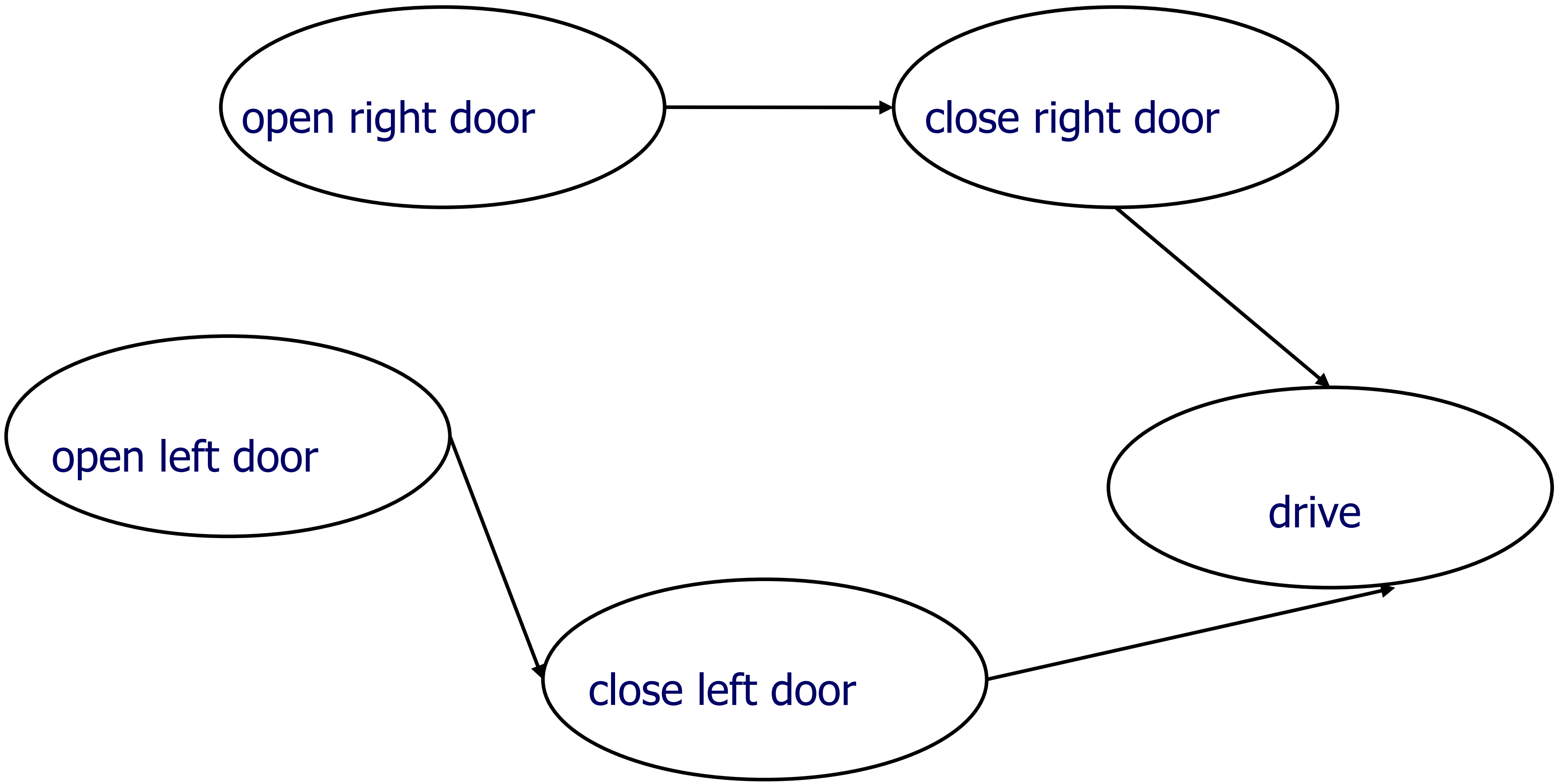
Overlaying a list on a dag

14.4.1 TOPOLOGIC SORTING OF DAGS

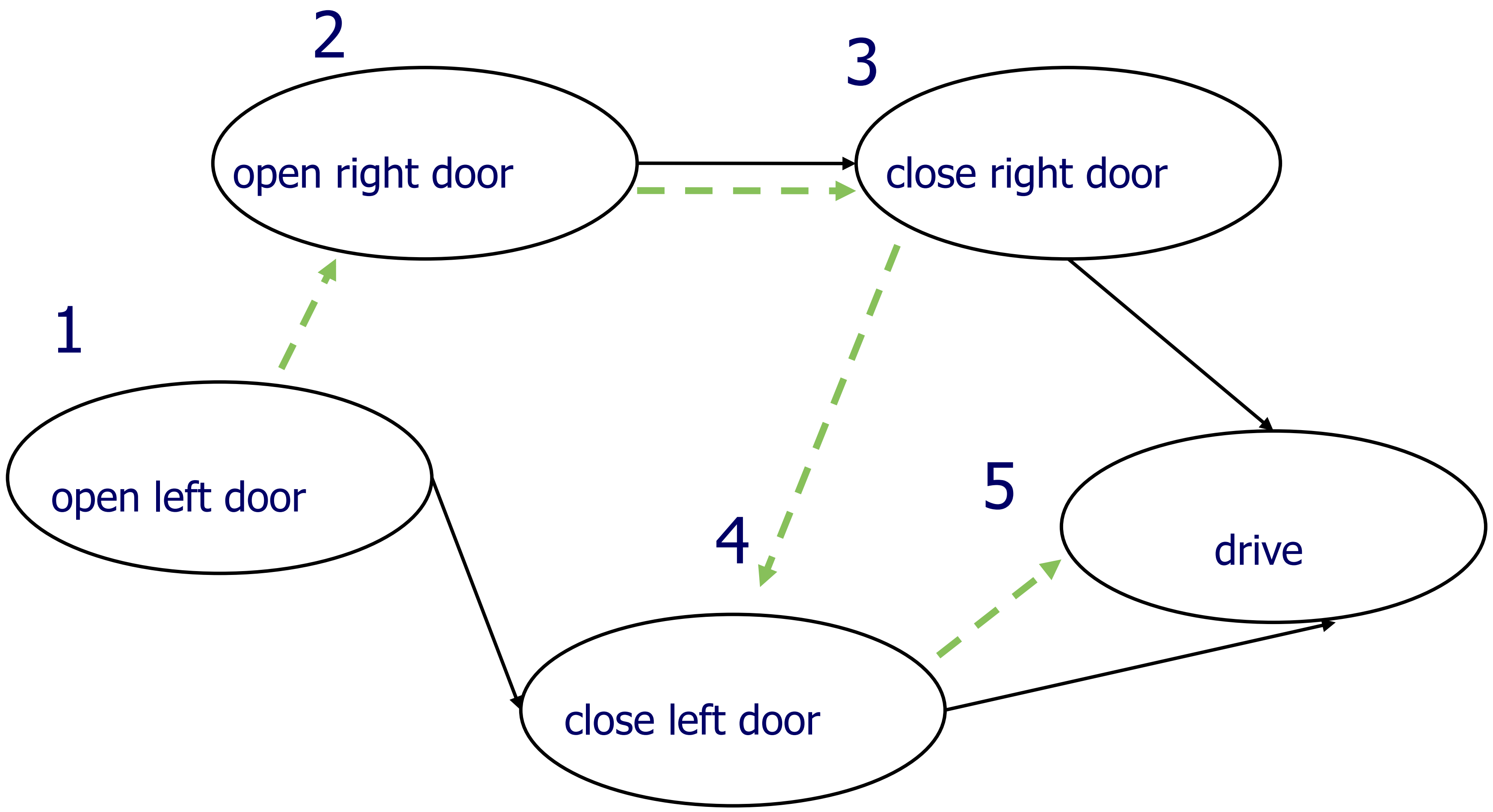
- If constraints for the partial order of some things are given, but no total order
- It doesn't matter in which order some things are executed
  - May be even in parallel
- There are many "legal" orderings, the topological sortings (topsorts)
- Totalordnung finden

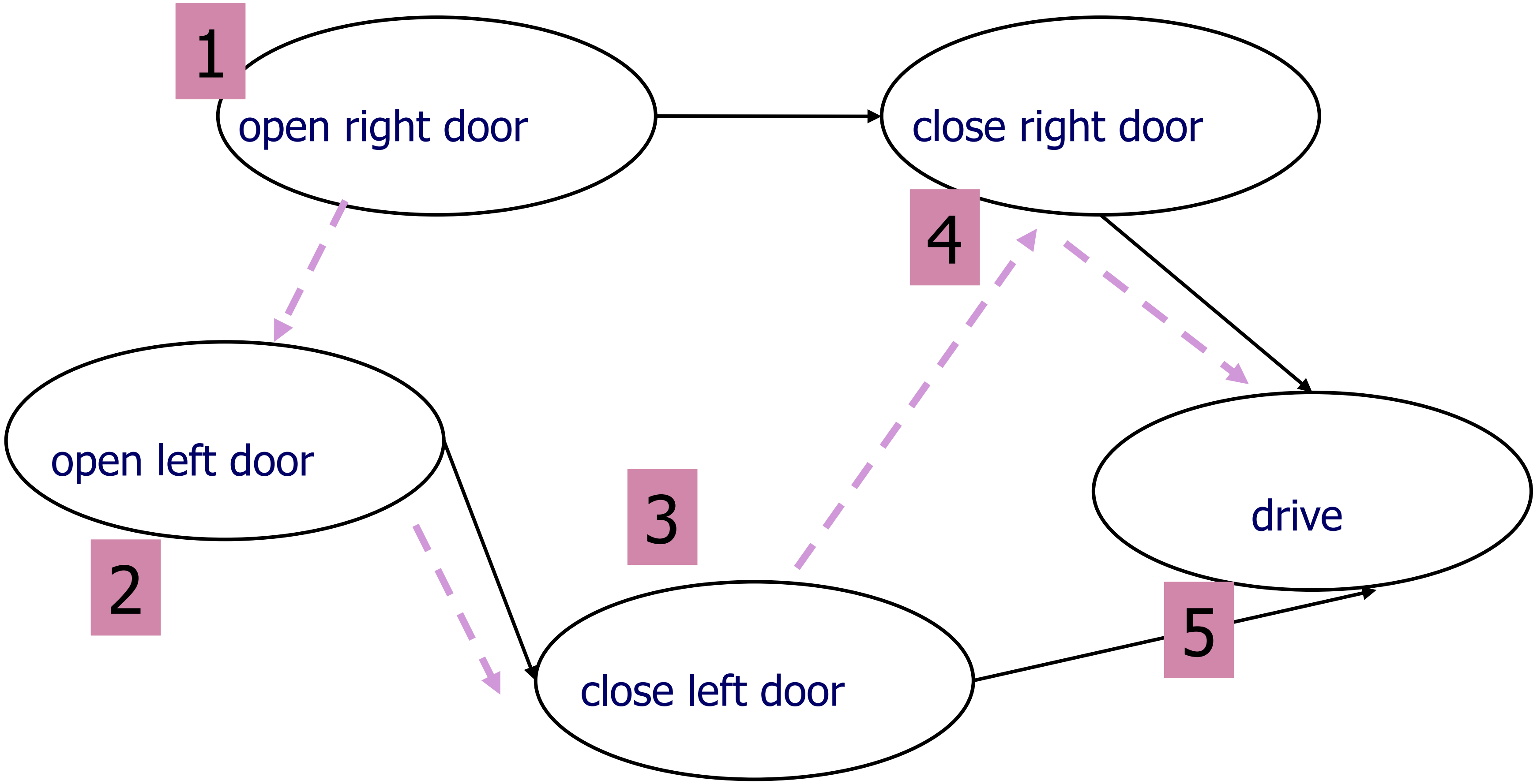


Partial Order for Car Departure



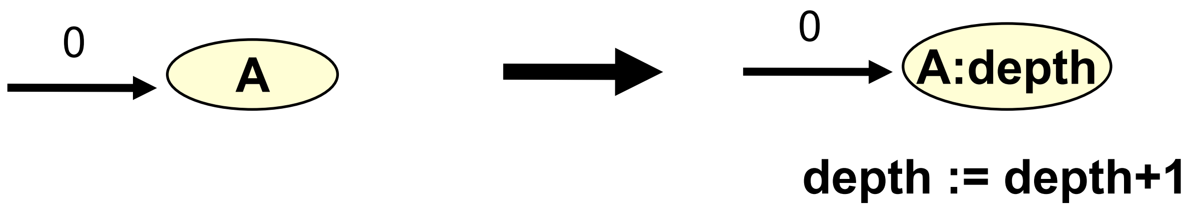
Topological Sorts on Car Departure



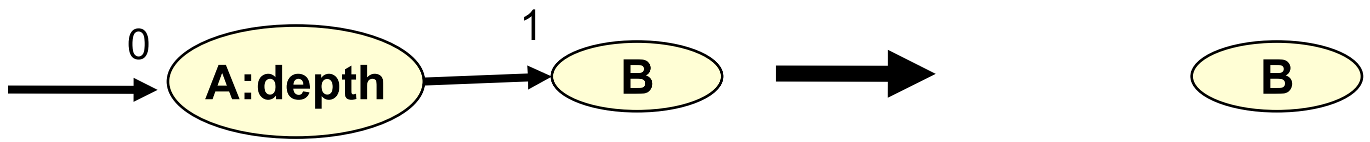

1
open right door
close right door
4
open left door
3
drive
2
5
close left door

- Topological sorting sorts the nodes with the „least many ancestors" first
- Described by a automatic graph rewrite system (SGRS)

R1: Numbering entry nodes

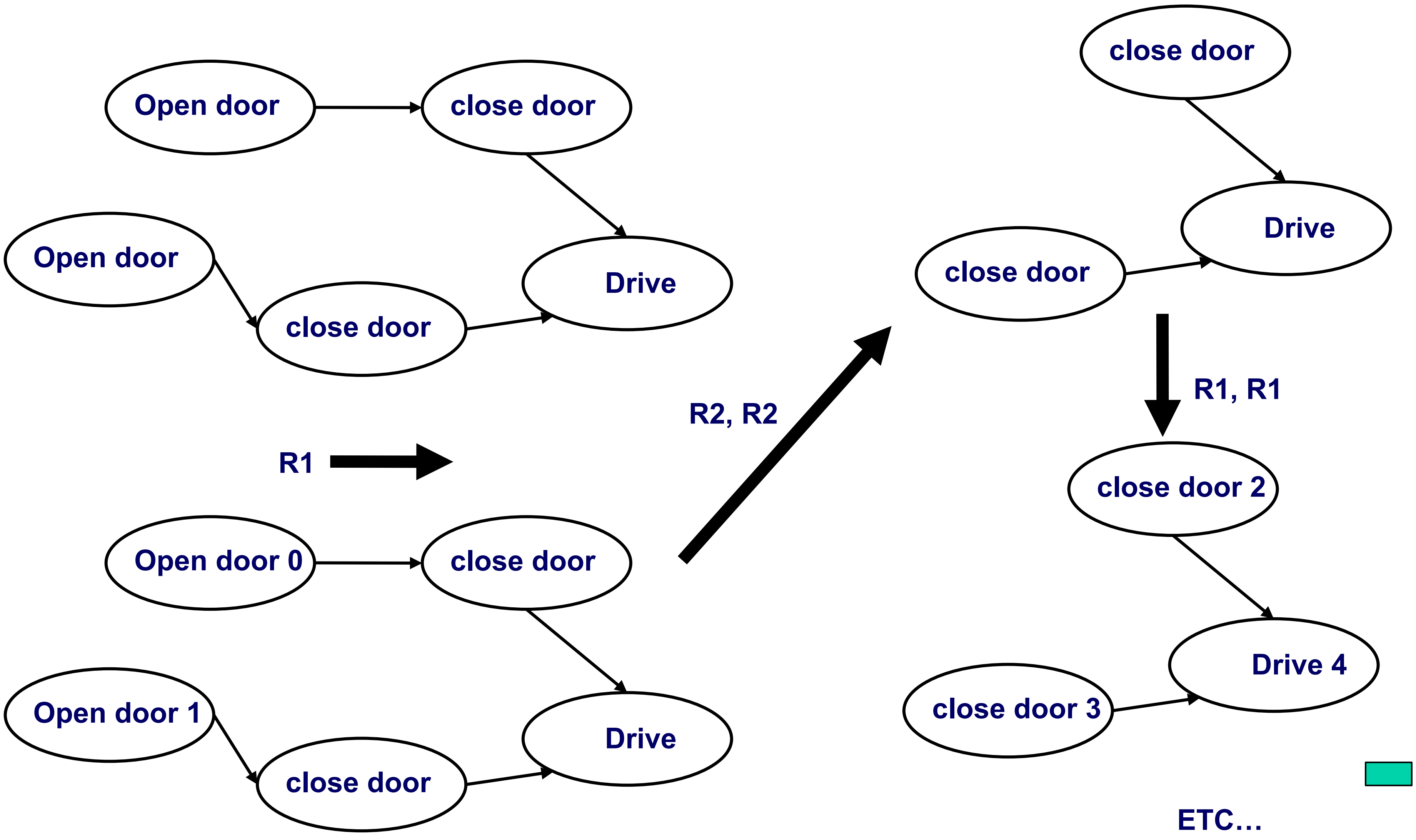


R2: Remove entry nodes

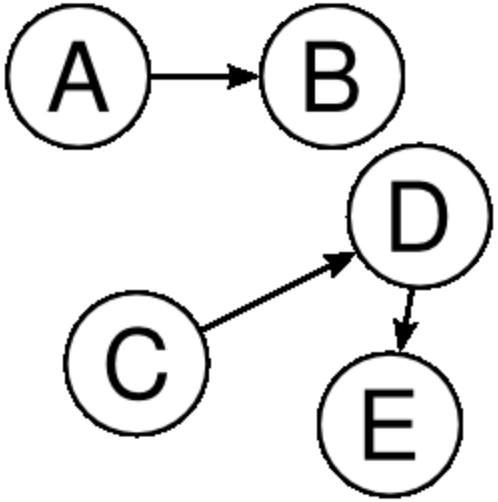


http://de.wikipedia.org/wiki/Topologische_Sortierung

Topological Sorts on Car Departure

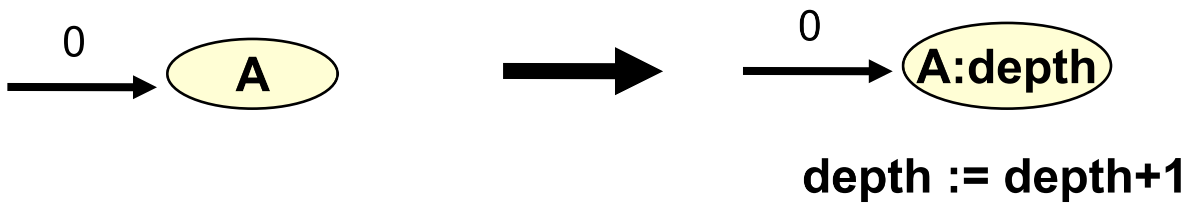


- AC BDE

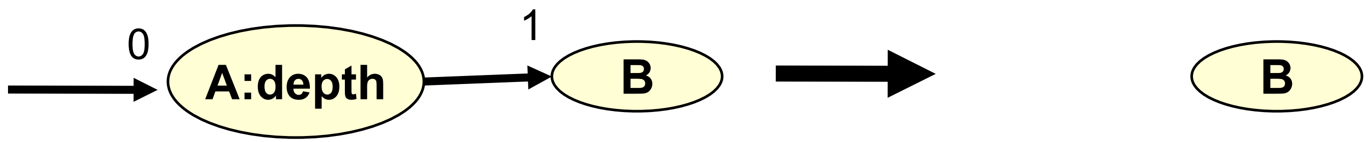


- Topological sorting sorts the nodes with the „least many ancestors" first.

TopSort-R1: Numbering entry nodes



TopSort-R2: Remove entry nodes

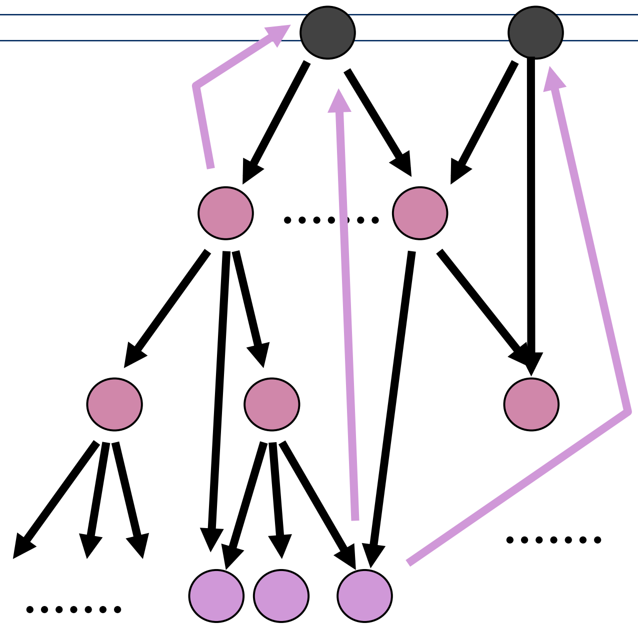


Benefit of TopSorts

- 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!



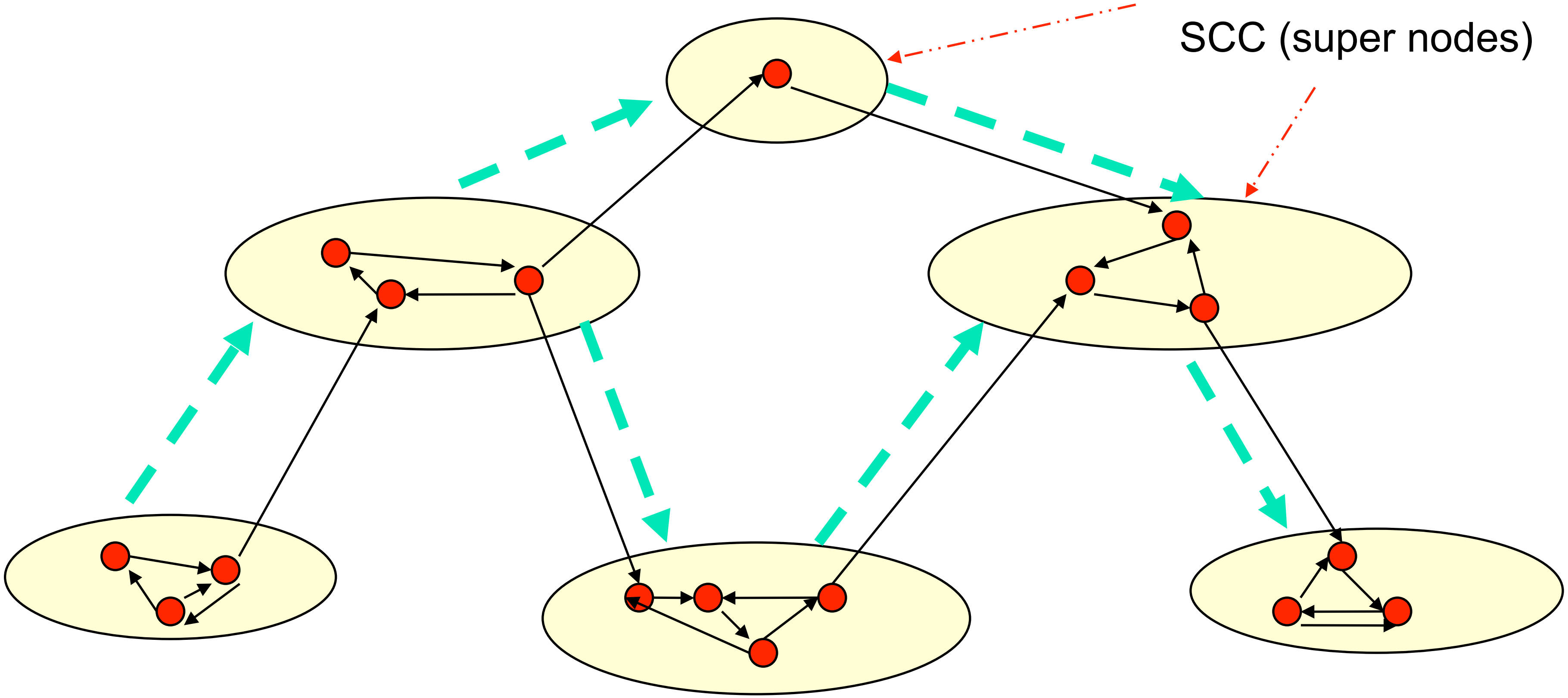
How to make an arbitrary relationship acyclic: overlaying a graph with a dag

14.4.2 STRONGLY CONNECTED COMPONENTS

Strongly Connected Components (Acyclic Condensation)

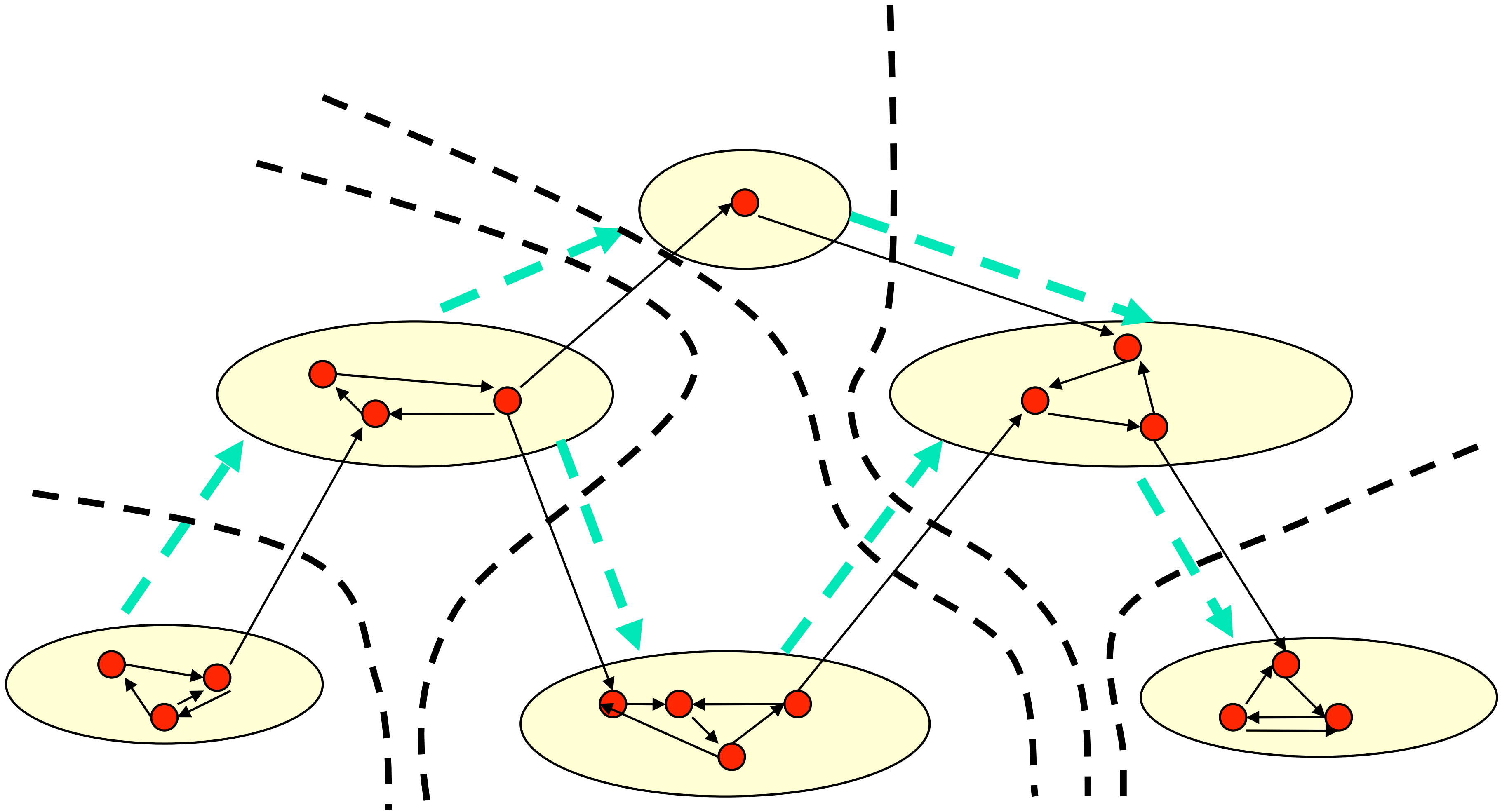
- The acyclic condensation asks for mutual reachability of nodes, hence for the effect of cycles in graphs
- A digraph is strongly connected, if every node is reachable from another one
- A subgraph of a graph is a strongly connected component (SCC)
  - If every of its nodes is strongly connected
- The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

- The SCC of a graph form „abstract super nodes"
- That digraph 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

A Wavefront on an AC



- SCCs can be made on every graph
  - Always a good structuring means for every kind of diagram in design
  - SCCs form “centers”
  - AC can always be topsorted, i.e., evaluated in a total order that respects the dependencies
- Useful for structuring large statecharts and Petri nets
  - Coalesce loops into subdiagrams
- Wavefronts
  - Analyzing statistics on graphs

Applications of SCC

- 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 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)

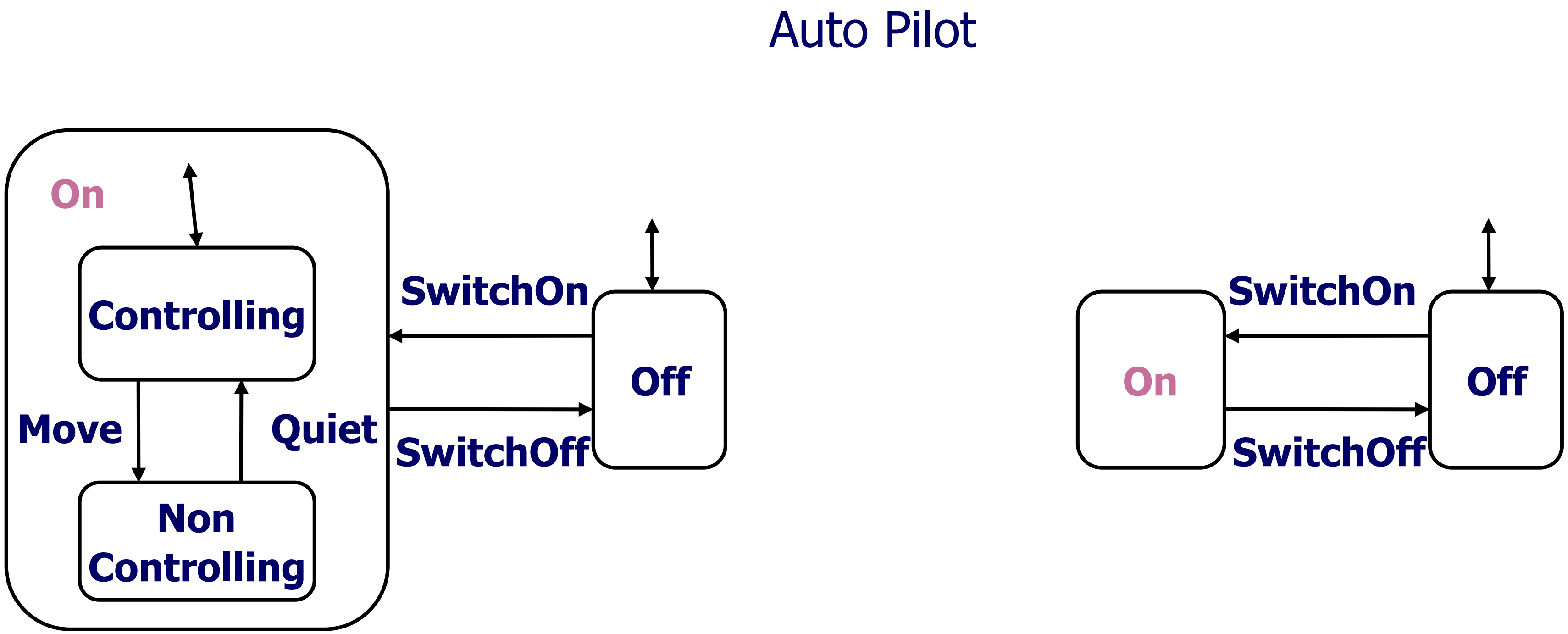
Has the graph a skeleton tree structure?

(Finding a hierarchy in a graph-based model)

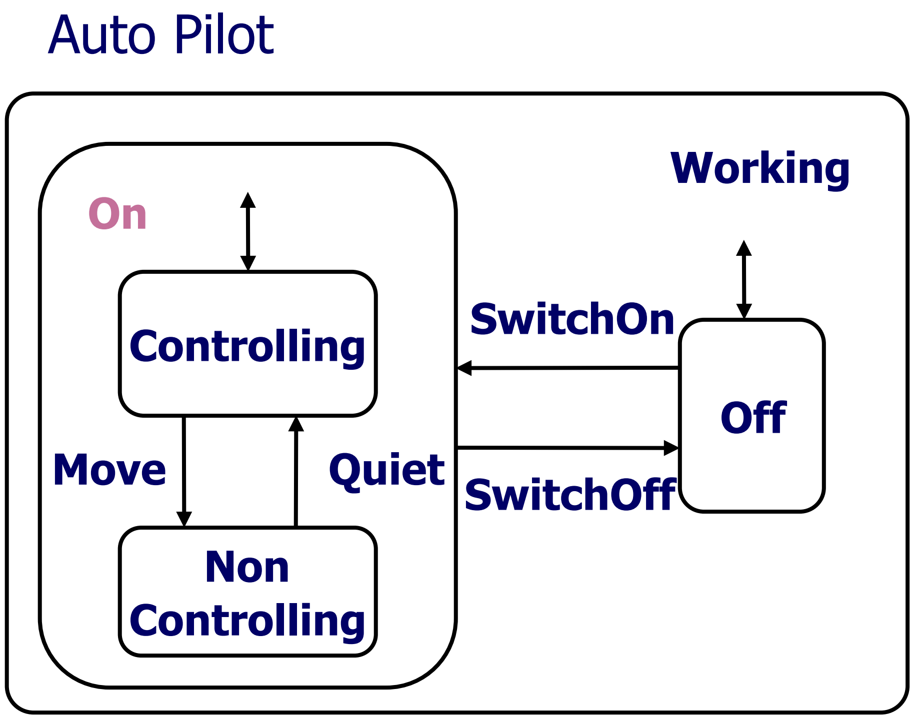
14.4.3 REDUCIBILITY

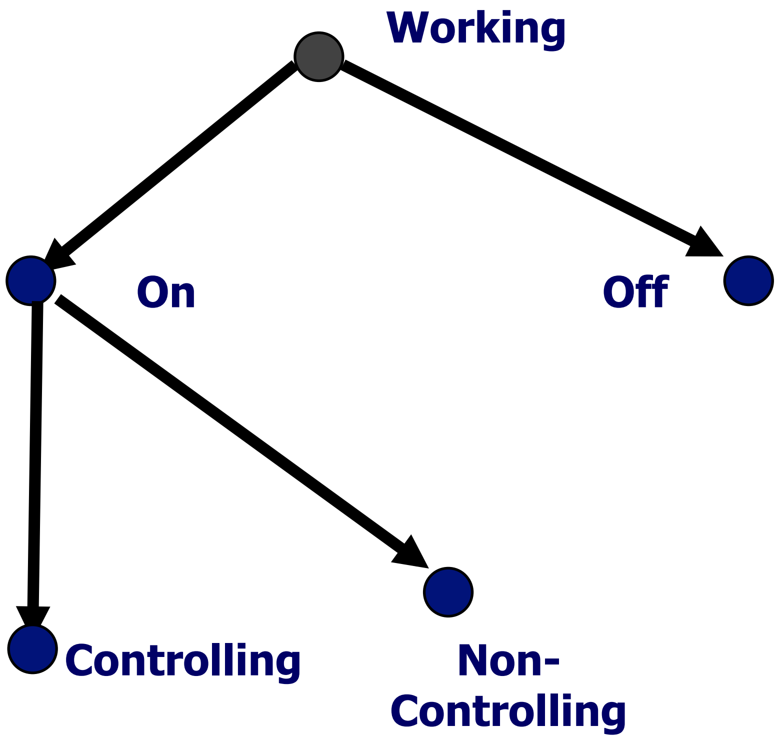
Why Is a UML Statechart Simple to Understand?

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



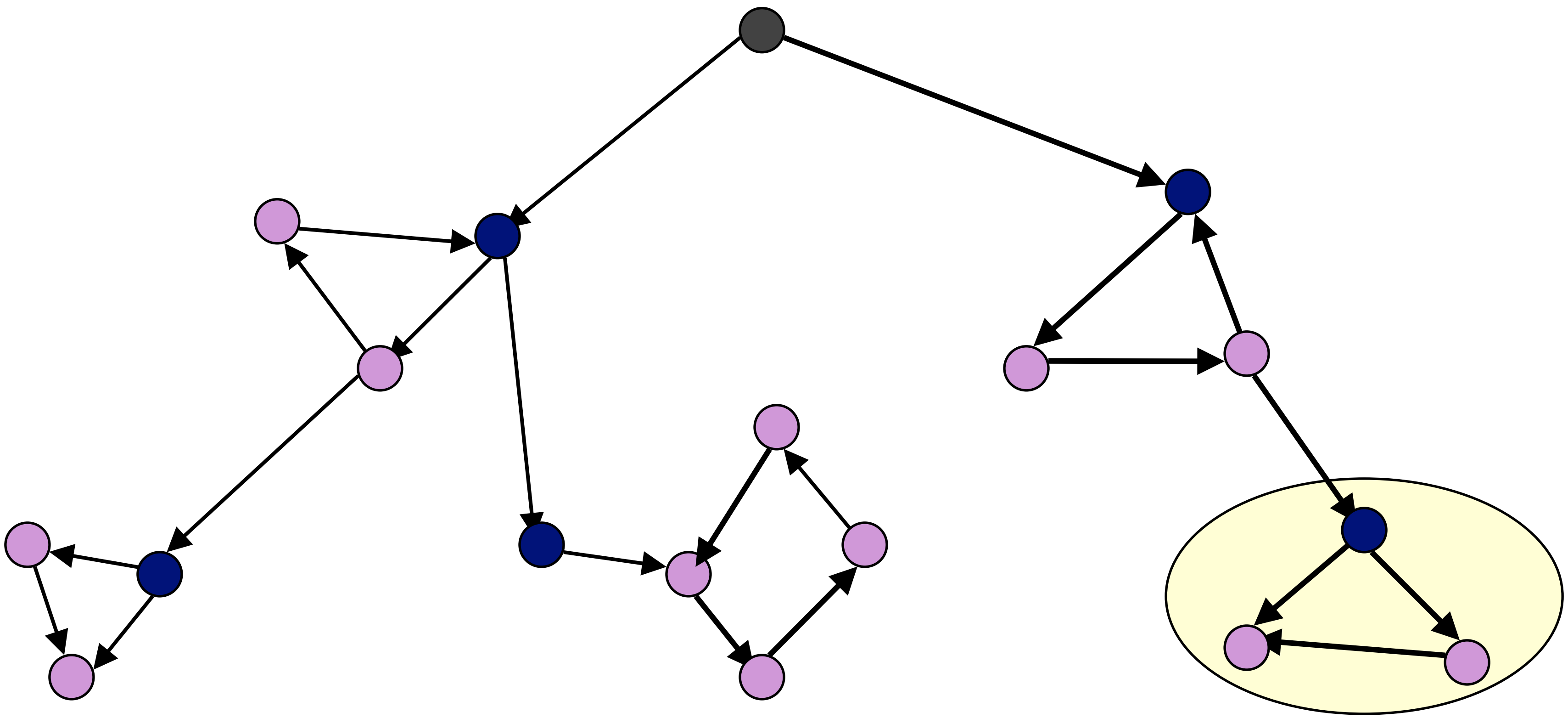
- But hierarchically organized



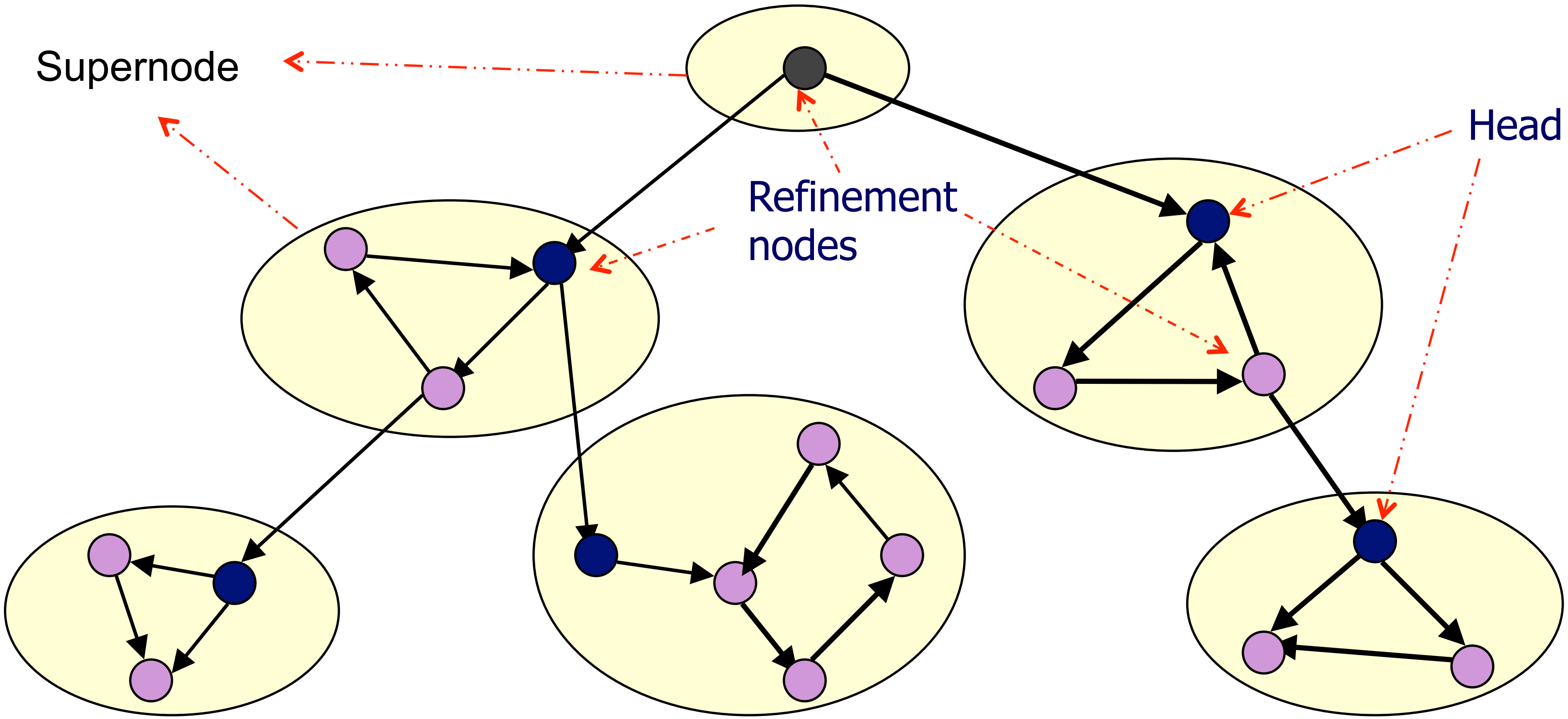


A Reducible Graph

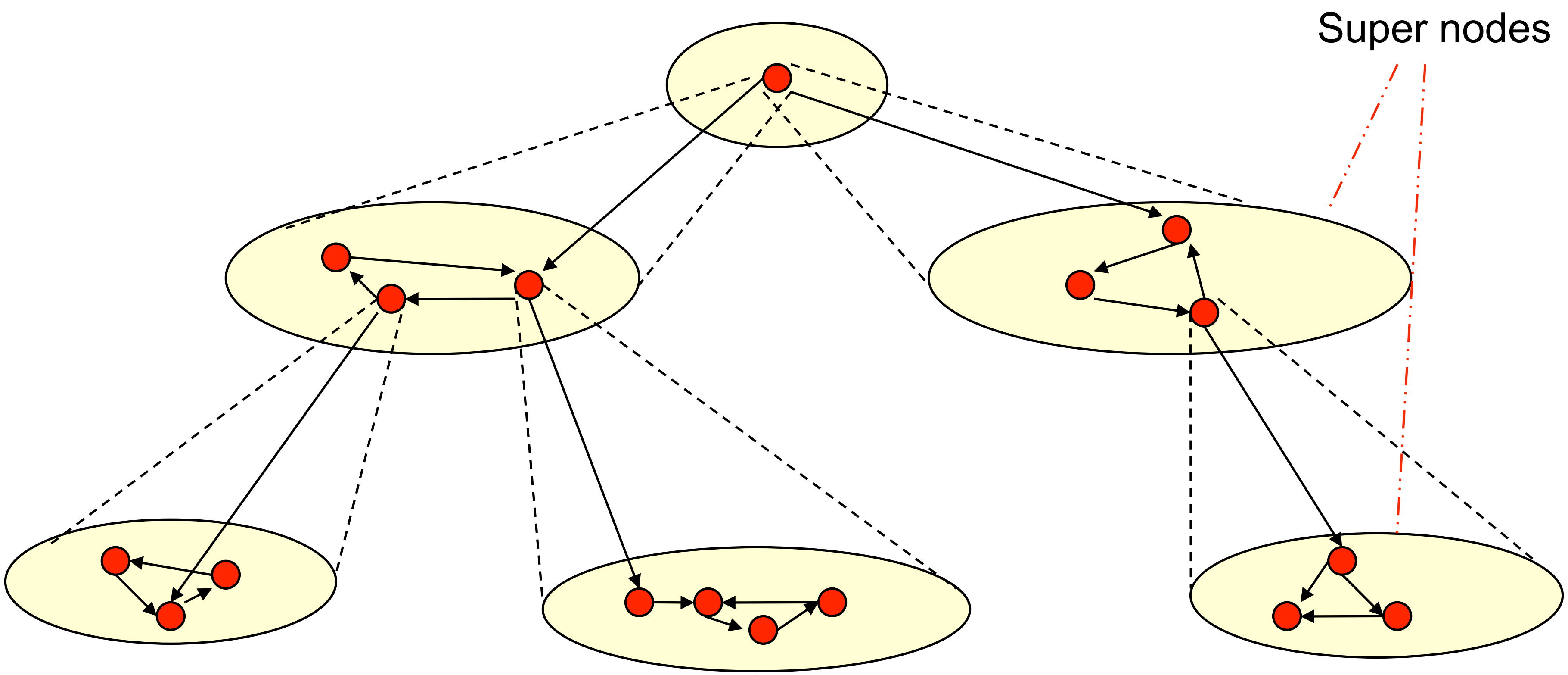
- A reducible graph has special areas with subdags and cycles, *supernodes*
- Attention: this is not an acyclic condensation!



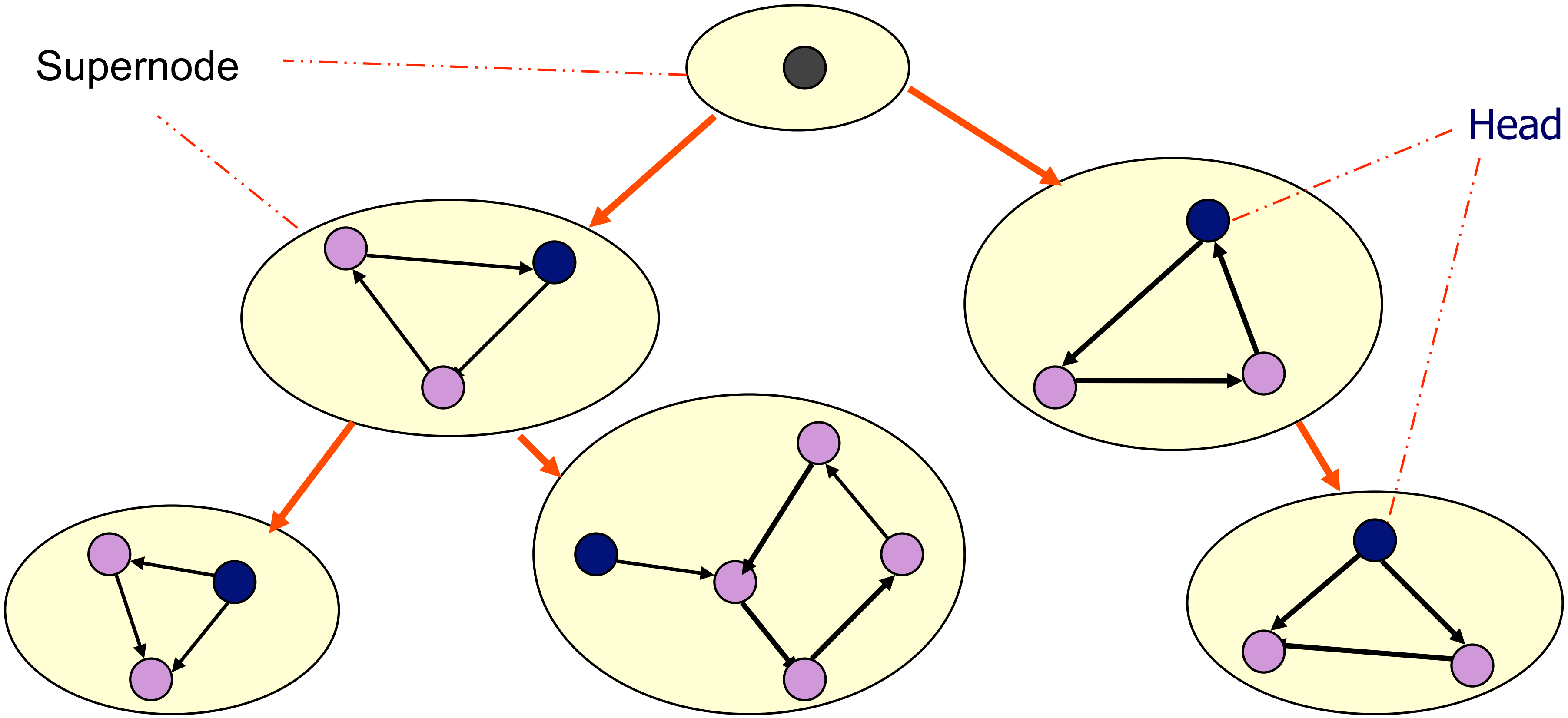
- Every supernode has a *head* that *represents* or *abstracts* it
  - All ingoing edges into the super node end in the head
  - The supernode is refined from a *refinement node*



- Reducible graphs have a hierarchical structure
  - A skeleton tree of super nodes with head nodes
  - Supernodes can hide subgraphs
  - Attention: SCC have a DAG structure (different!)



- A *skeleton tree* between the supernodes results
- Much simpler!



Reducible Graphs in Software Engineering

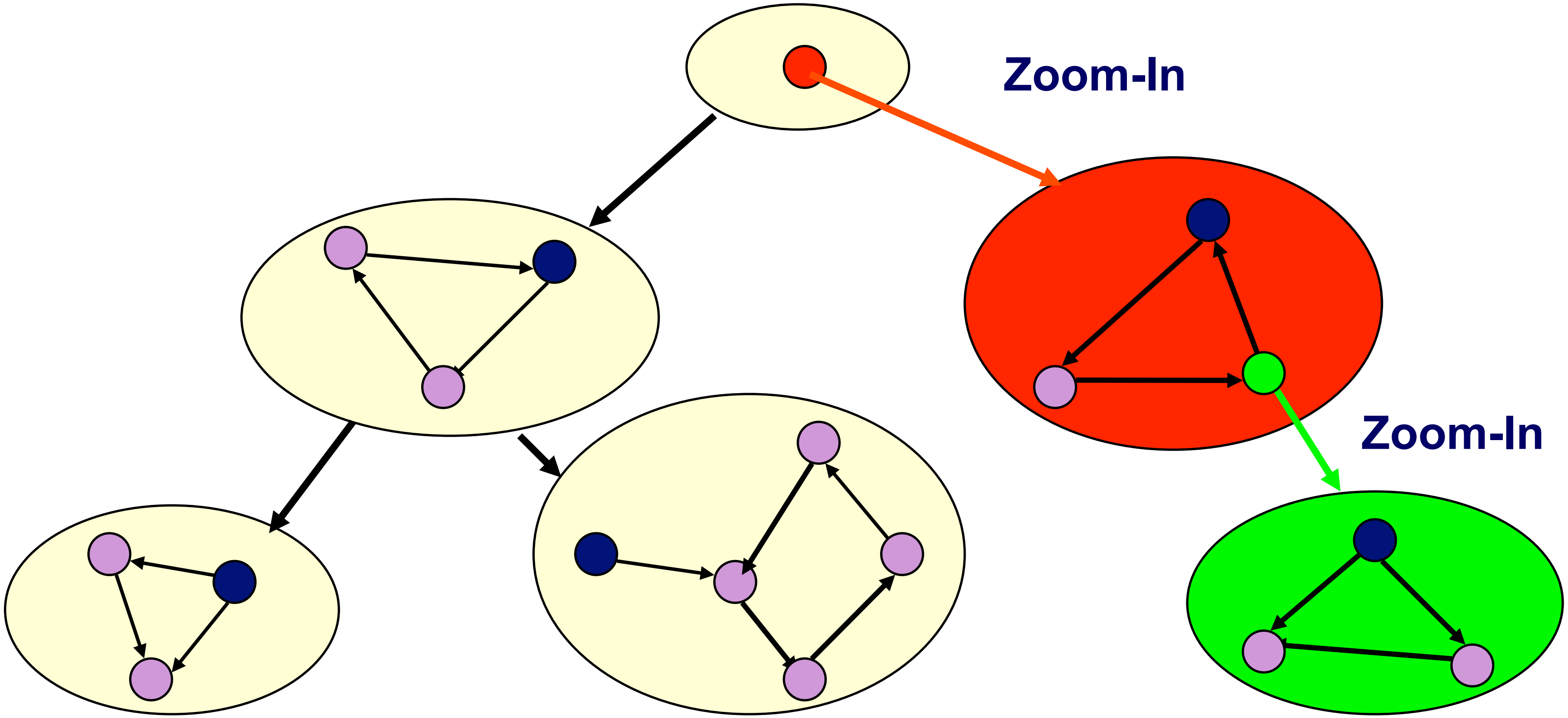
- Submodels can be *abstracted* into single nodes
- Whole model can be abstracted into one node
- Skeleton tree structures the model

> A model *should* use reducible graphs

- Otherwise large models cannot be understood

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*



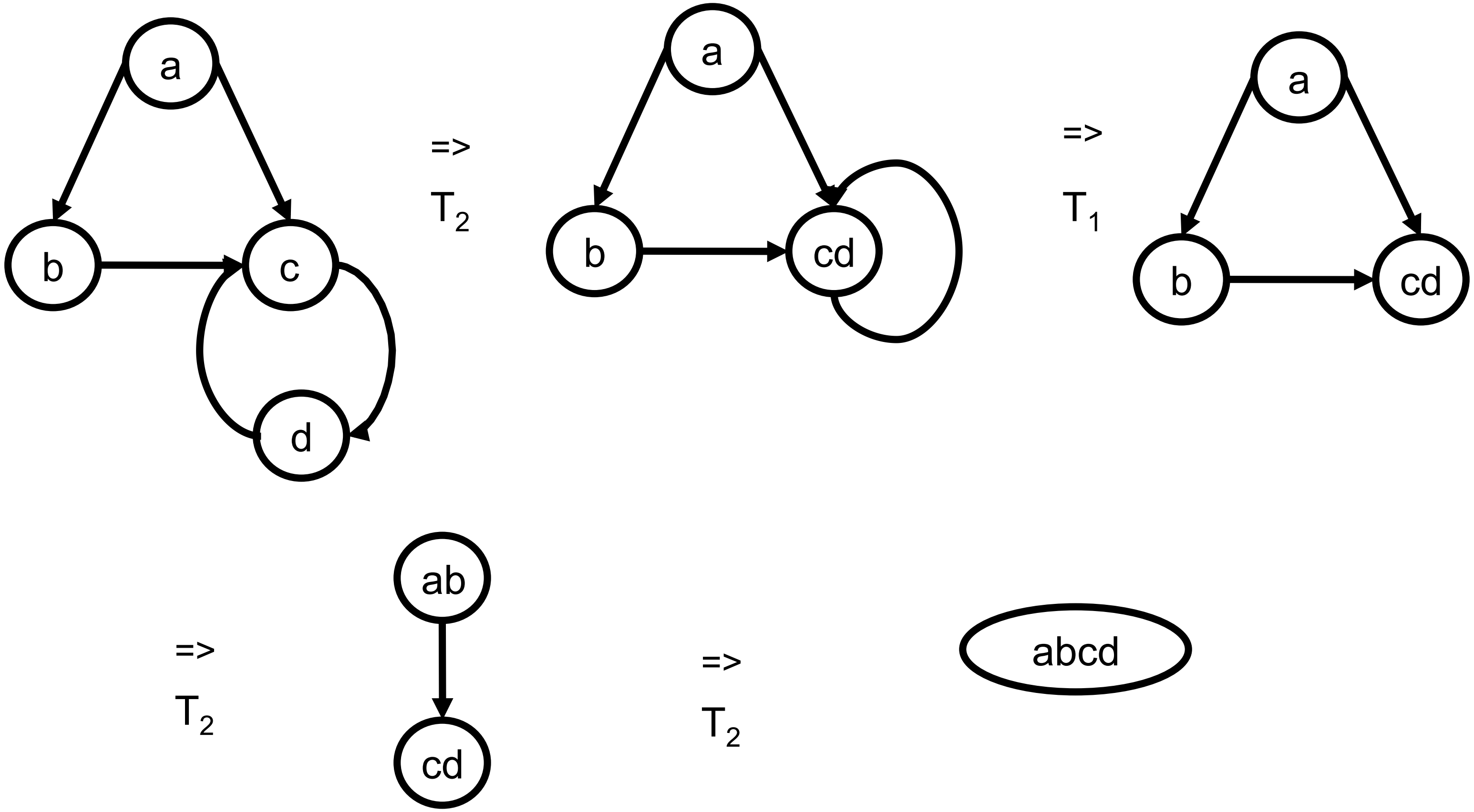
- A *reducible* digraph is a digraph, that can be reduced to one node by the following graph rewrite rules
- Specification with a automatic GRS (SGRS):

T1: Remove reflective edges

T2: Merge successors with fan-in 1

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

- On every level, in the super nodes there may be cycles
  - T2 shortens these cycles
  - T1 reduces reflective cycles to super nodes



Reducible Graphs

- All recursion techniques on trees can be taken over to the skeleton trees of the reducible graphs
- 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

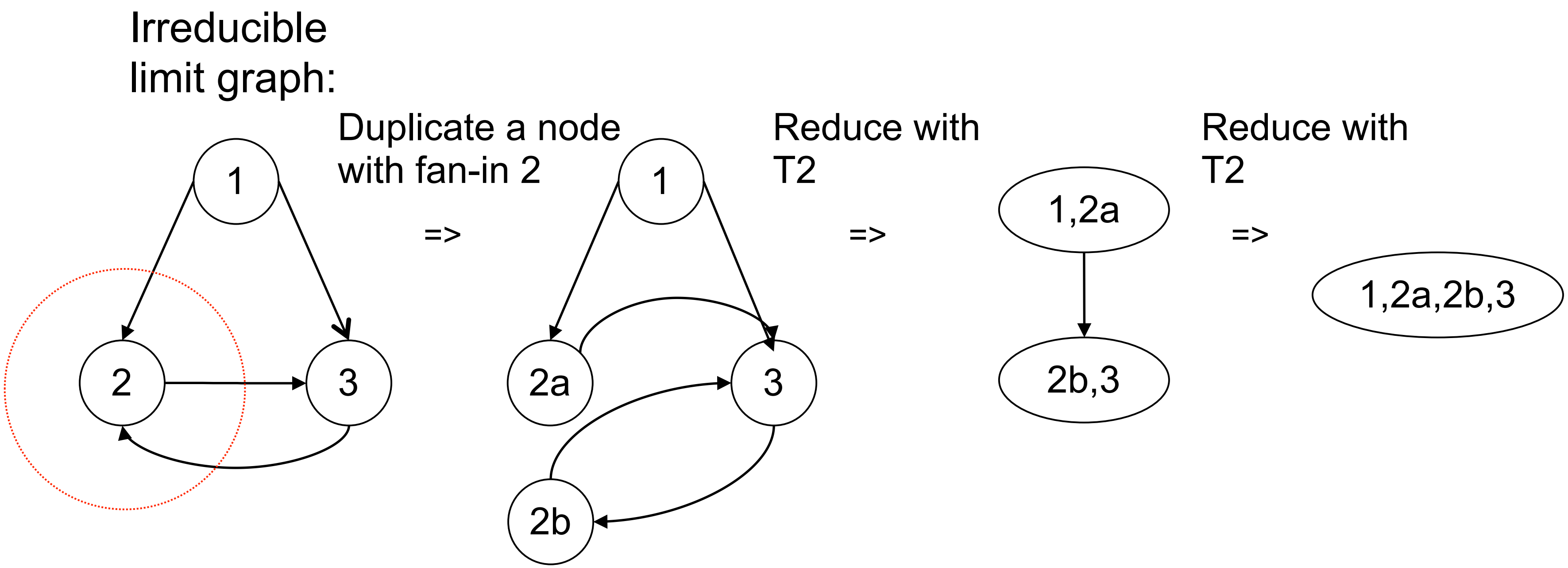
- Structure UML Class Diagrams
- Choose an arbitrary UML class diagram
- Calculate reducibility
  - If the specification is reducible, it can be collapsed into one class
  - Reducibility structure gives a simple package structure
- Test dag feature
  - If the diagram is a dag, it can be layered
- TopSort 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

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.

- Duplicate one node in an irreducible loop (even with subtrees)



14.5 SUMMARY OF STRUCTURINGS

- Structurings Producing Lists
- Layering
  - Overlaying a list of layers onto a dag
  - "same generation problem"
  - Standard Datalog, DL, EARS problem
- Topological sorting
  - Overlaying a dag with a list
  - Totalizing partial orders
  - Solved by a graph rewrite system
  - Sequentializing parallel systems

- **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

	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