TECHNISCHE UNIVERSITÄT DRESDEN

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

14. How to Transform Models with Graph Rewriting

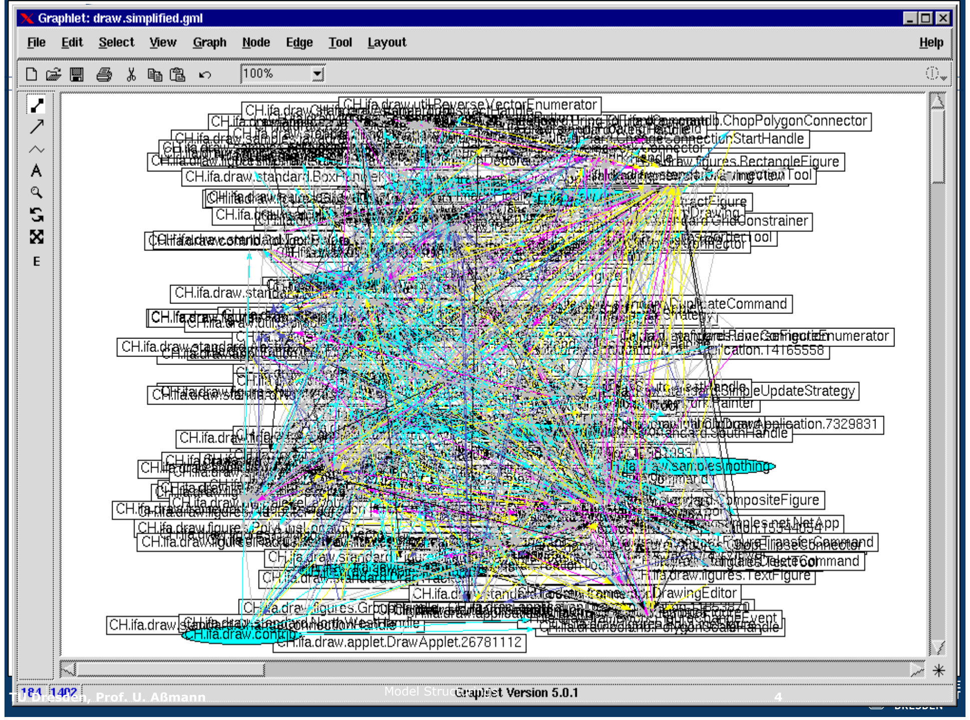
Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Gruppe Softwaretechnologie
http://st.inf.tu-dresden.de
Version 12-1.0, 21.11.12

Obligatory Reading

- Jazayeri Chap 3. If you have other books, read the lecture slides carefully and do the exercise sheets
- T. Mens. On the Use of Graph Transformations for Model Refactorings. In GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/
- 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/
- 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
  - [ASU86] Alfred V. 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

14.1 GRAPH TRANSFORMATIONS

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

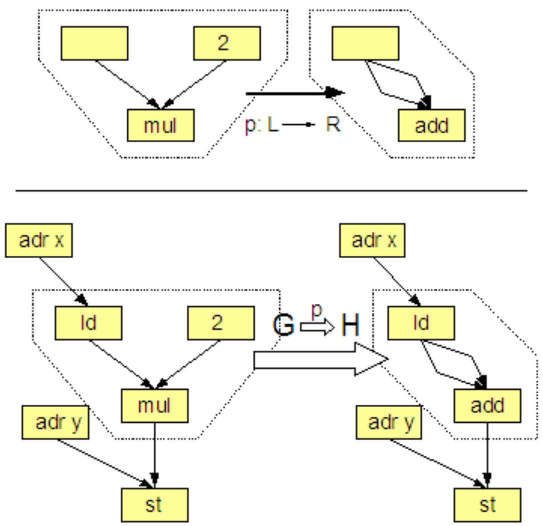
Graph Rewrite Systems

- 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

GRS Example

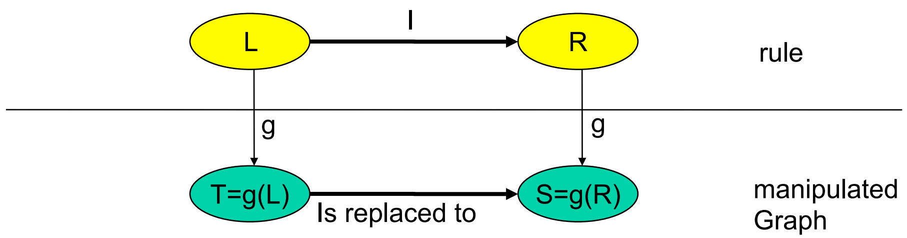
Rule

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)

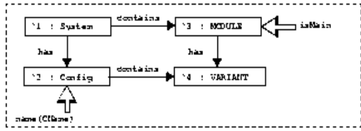


PROGRES, the GRS tool from the IPSEN Project

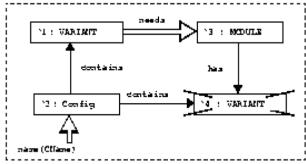
- PROGRES is a wonderful tool to model graph algorithms by graph rewriting
- Textual and graphical editing
- Code generation in several languages

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

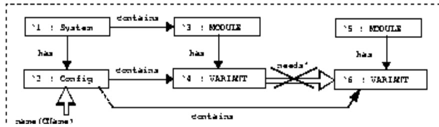

```
query ConsistentConfiguration( out CName : string ) =

   (* 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 ) =
```

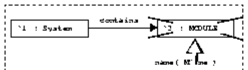



```
end;

test ConfigurationWithUselessVariant( CName : string ) =
```

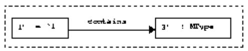


This example illustrates the possibilities of PROGRES to define *parametrized productions* which must be instantiated (in the sense of a procedure call) with actual attribute values and node types. In this way, a single production may abstract from a set of productions which differ only with respect 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.

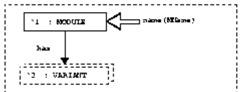
```
production CreateModule ( MName : string; InterfaceDescription : file;
                          MType : type in MODULE ) =
```




```
   ::=
```

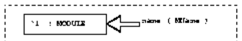



```
   transfer 2'.Name := MName;
            2'.File := InterfaceDescription;
end;

production DeleteModule ( MName : string ) =
```

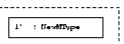



```
   ::=
```

```
end;

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




```
   ::=
```

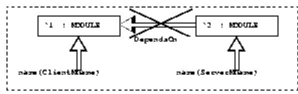



```
   embedding redirect  -m_uses->, -has->, <-contains-,
                       <-m_uses-, <-v_uses- from '1 to 1';
   transfer 1'.Name := '1.Name;
            1'.File := '1.File;
end;
```

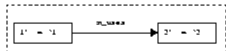
Fig. 12: Specification of basic graph transformations

-20-

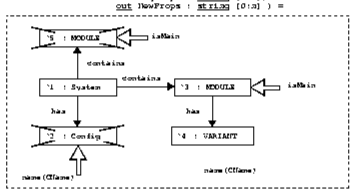

```
production CreateMUse( ClientMName, ServerMName : string ) =
```



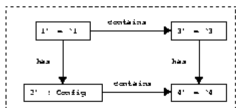

```
  ::=
```




```
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-

Type Scheme of a Graph

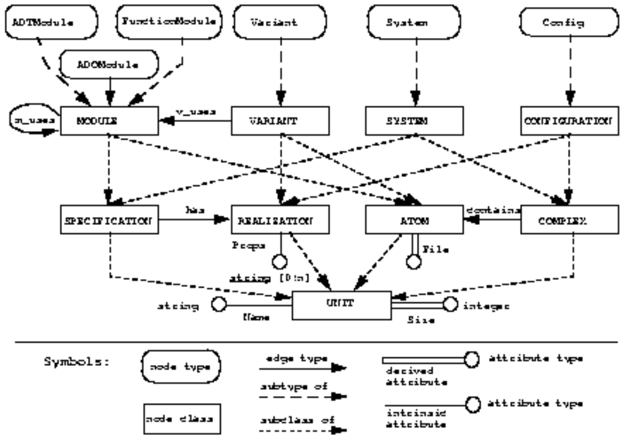


Fig. 5: The graph schema of MIL graphs (without derived relationships)

- *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

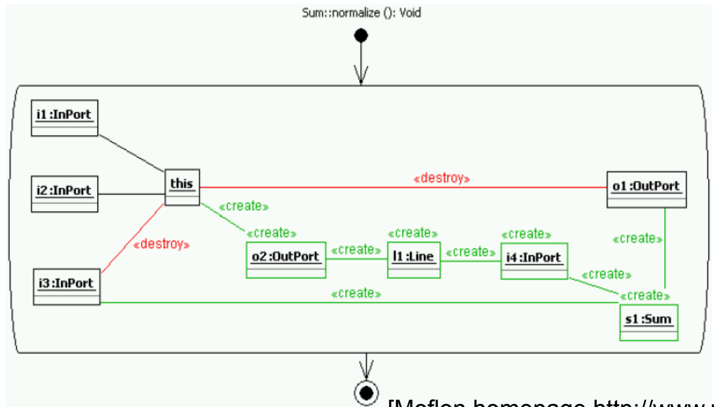
- Automatic Graph Rewriting
  - Iteration of rules until termination
- Programmed Graph Rewriting
  - The rules are applied of a control flow program. This program guarantees termination and selects one of several solutions
  - Examples: PROGRES from Aachen/München
  - Fujaba on UML class graphs, from 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)

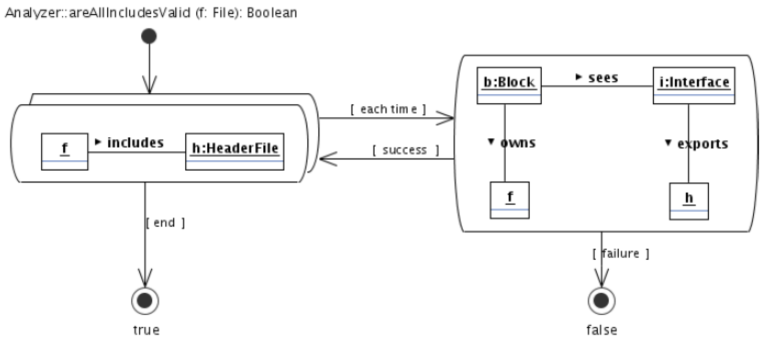
MOFLON and Fujaba

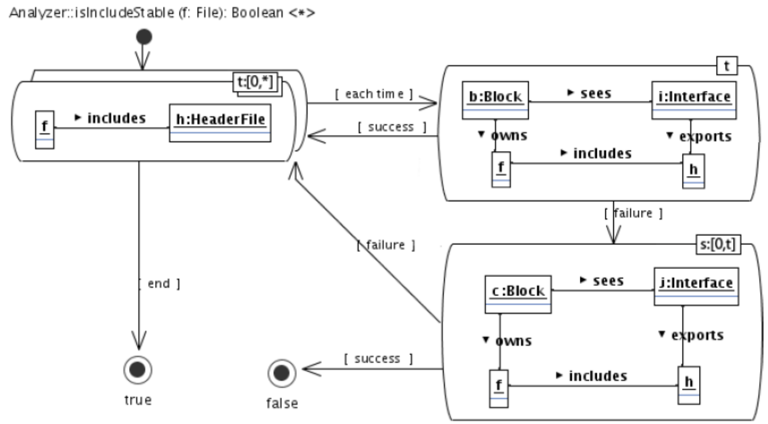
- MOFLON and Fujaba embed graph rewrite rules into activity diagrams (aka storyboards)
  - A rule set executes as an atomic activity
  - Colors express actions



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

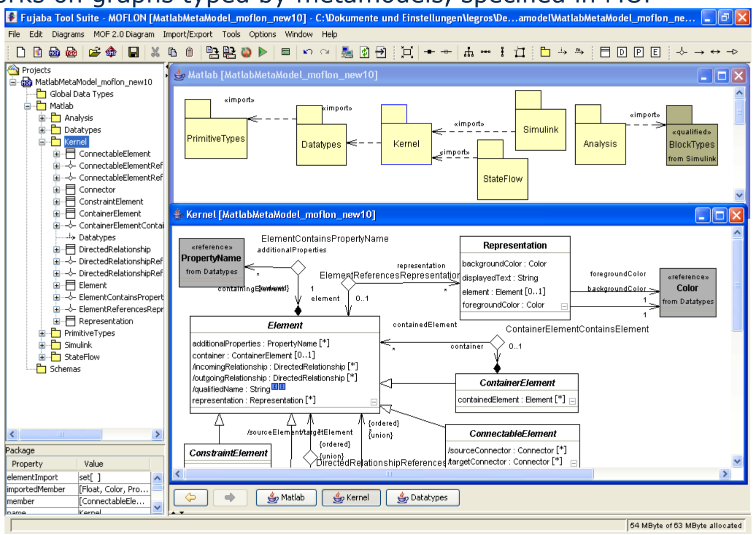
Storyboards are Refined Activity Diagrams





MOFLON

- Works on graphs typed by metamodels, specified in MOF



Mapping graphs to other graphs

Specification of mappings with mapping rules

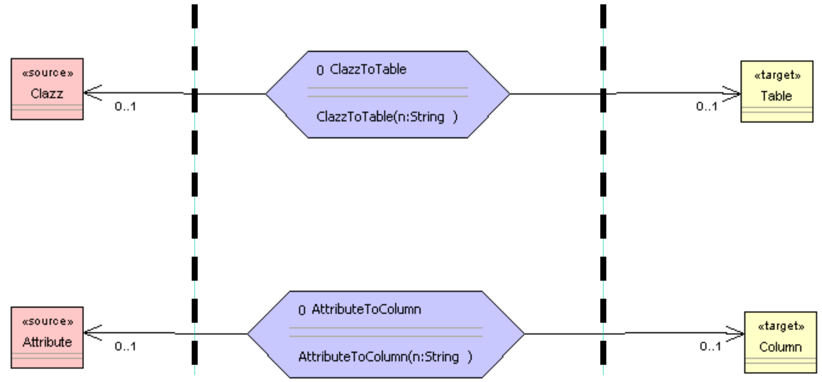
Incremental transformation

Traceability

14.3 „SYNCHRONIZING" MODELS WITH TRIPLE GRAPH GRAMMARS

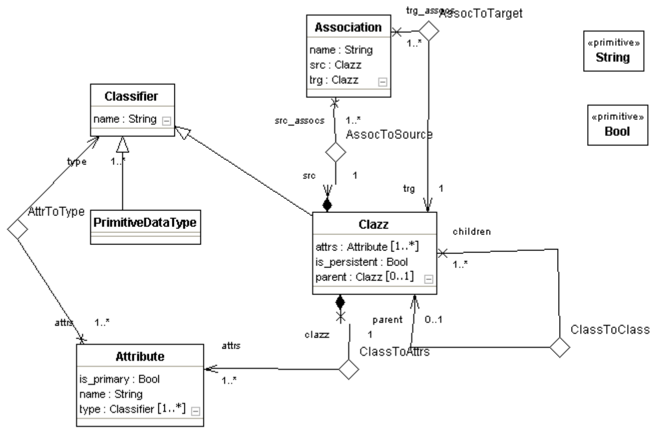
Triple Graph Grammars – Moflon Example

- A triple Graph Grammar (TGG) consists of rules with three areas"
  - Left side: graph pattern 1 in graph 1
  - Right side: graph pattern 2 in graph 2
  - Middle: relational expression (net) relating graph pattern 1 and 2



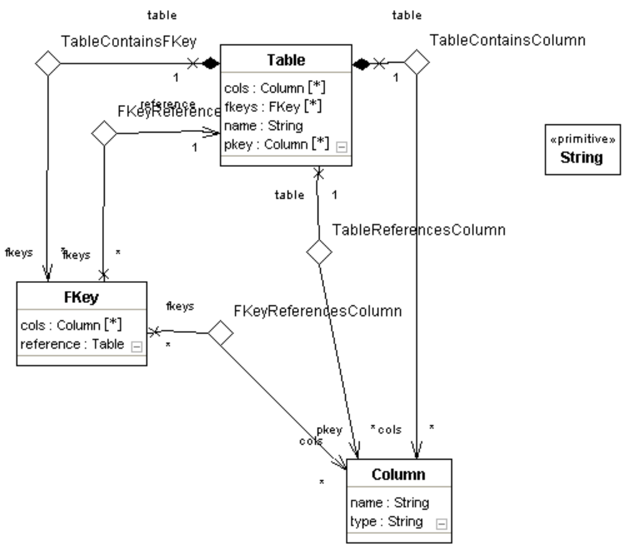
Triple Graph Grammars – Moflon Example

- Synchronize object-metamodel with a relational schema (ORM)
- Class diagram metamodel (CD)



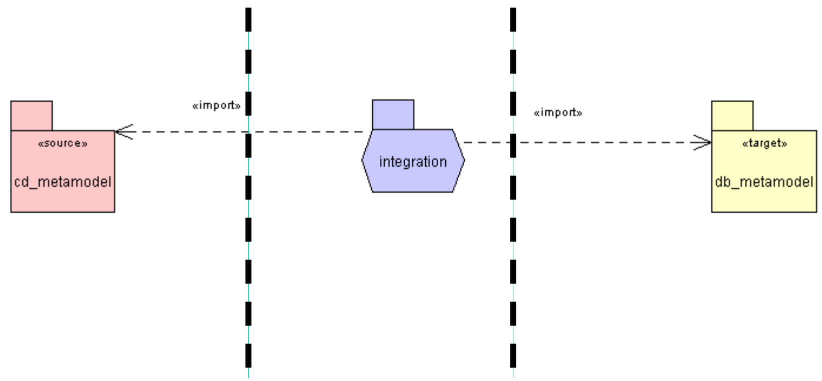
Triple Graph Grammars – Moflon Example

- Relational metamodel (db, relational schema)



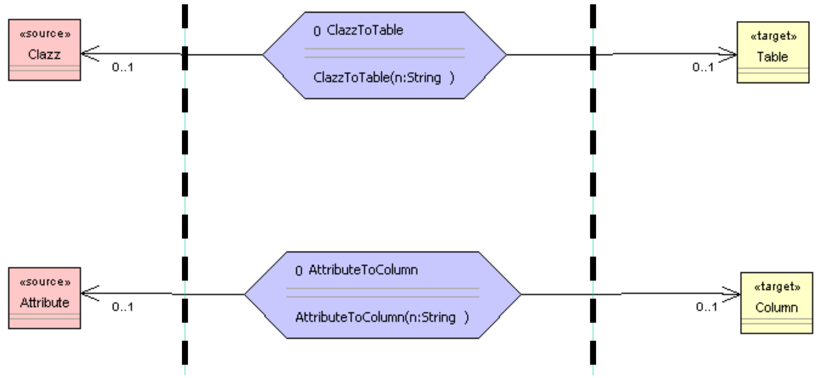
Triple Graph Grammars – Moflon Example

- A TGG has a top rule (start rule) which describes the relationship of the graphs on topmost level



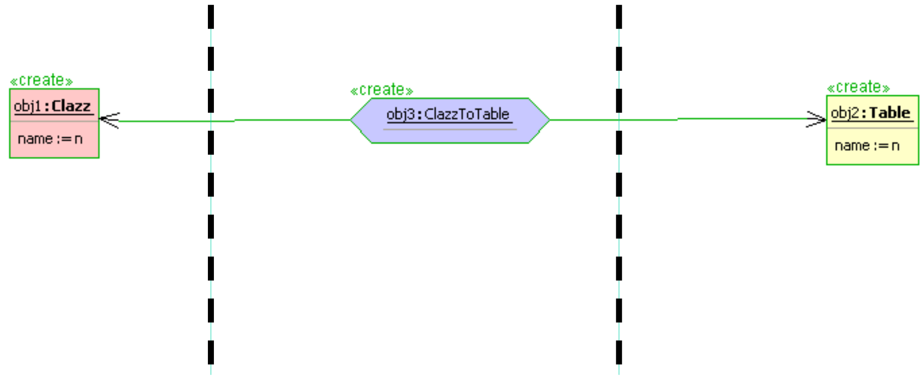
Triple Graph Grammars – Moflon Example

- From the top-rule, other TGG rules are „called"



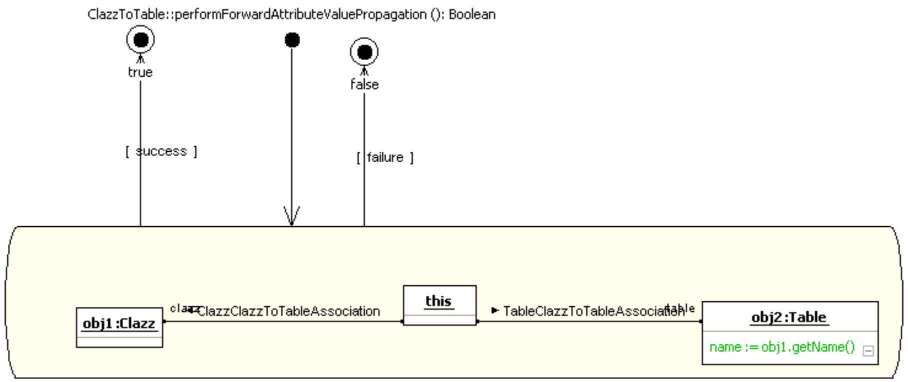
Triple Graph Grammars – Moflon Example

- This rule connects a class in the Object Model to the Table in the relational schema



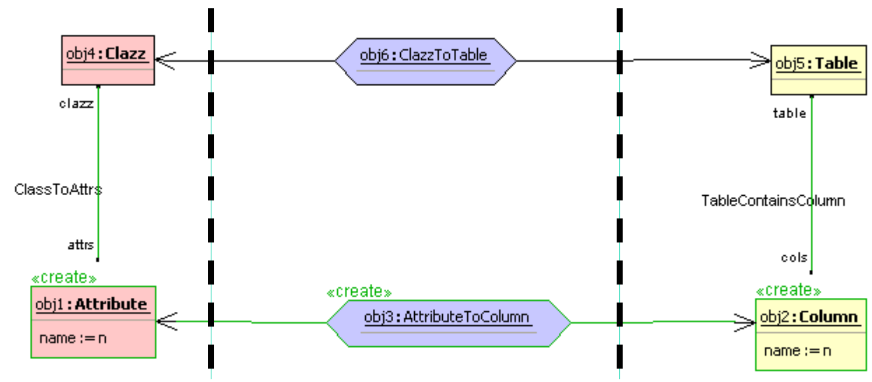
Triple Graph Grammars – Moflon Example

- TGG rules can be connected by Fujaba Storyboards

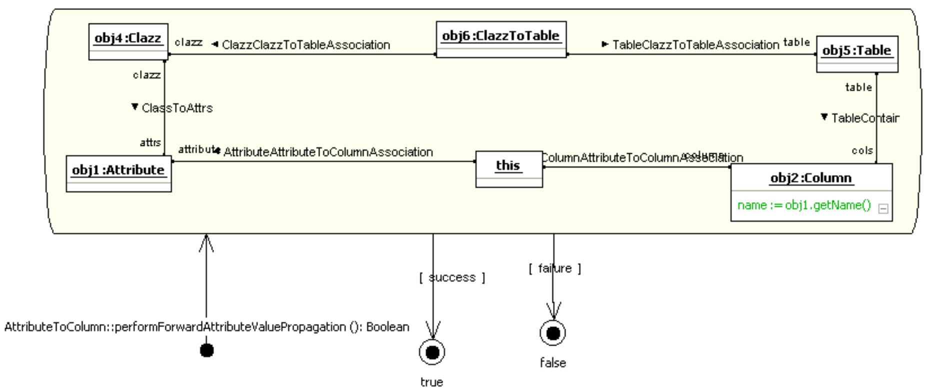


Triple Graph Grammars – Moflon Example

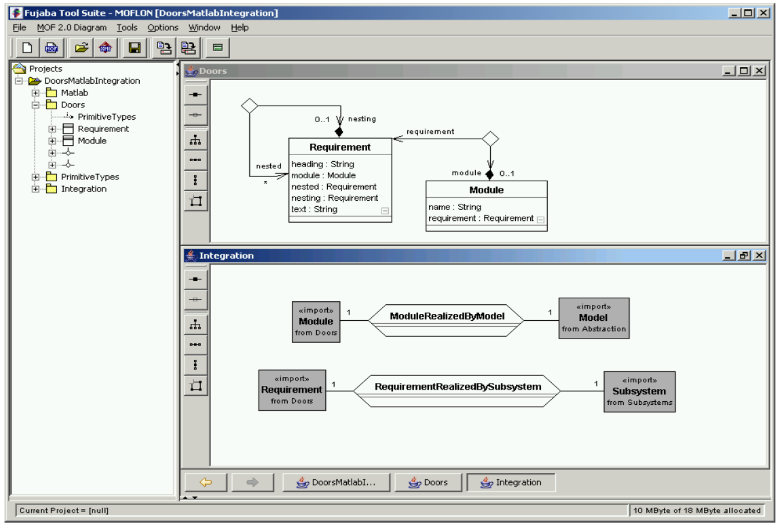
- Pairwise correspondance



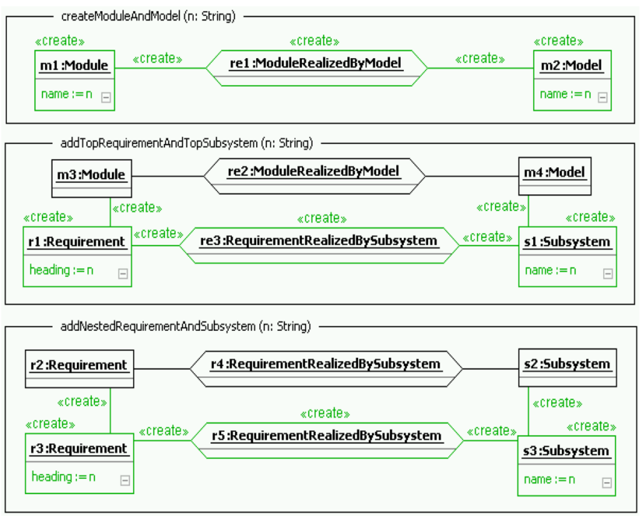
Triple Graph Grammars – Moflon Example



TGG Coupling Requirements Specification and Design



TGG Coupling Requirements Specification and Design



Other Software Engineering Applications

- Graph Structurings (see later)
- Refactorings (see Course DPF)
- Semantic refinements
- Round-Trip Engineering (RTE)

The End: What Have We Learned

- Graph rewrite systems are tools to transform graph-based models and graph-based program representations
- TGG enable to bidirectionally map models and synchronize them