24. Deep Analysis in Link-TreeWare (EMF and XML like)

EMF as Link-TreeWare

Prof. Dr. U. Aßmann

Technische Universität Dresden

Institut für Software- und Multimediatechnik

http://st.inf.tu-dresden.de/teaching/most

Version 21-1.1, 11.12.21

1) RAGs for link trees
2) Deep analysis with RAG of textual languages
3) Deep analysis of models with JastEMF
4) Using Querying as Attributions
5) Consequences for MDSD applications

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Obligatory Literature on RAG

- [Hedin11] An Introductory Tutorial on JastAdd Attribute Grammars. In Generative and Transformational Techniques in Software Engineering III, 6491:166-200. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.
  - https://link.springer.com/chapter/10.1007/978-3-642-18023-1_4
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.5911&rep=rep1&type=pdf
- [Bürger+11] Bürger, Christoff, Sven Karol, Christian Wende, und Uwe Aßmann. 2011. Reference Attribute Grammars for Metamodel Semantics. In Software Language Engineering. Springer Berlin / Heidelberg.
- [Heidenreich+12] Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert, und Christian Wende. 2012. „Model-based Language Engineering with EMFText“. In Generative and Transformational Techniques in Software Engineering, 7680:322ff. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Informative References

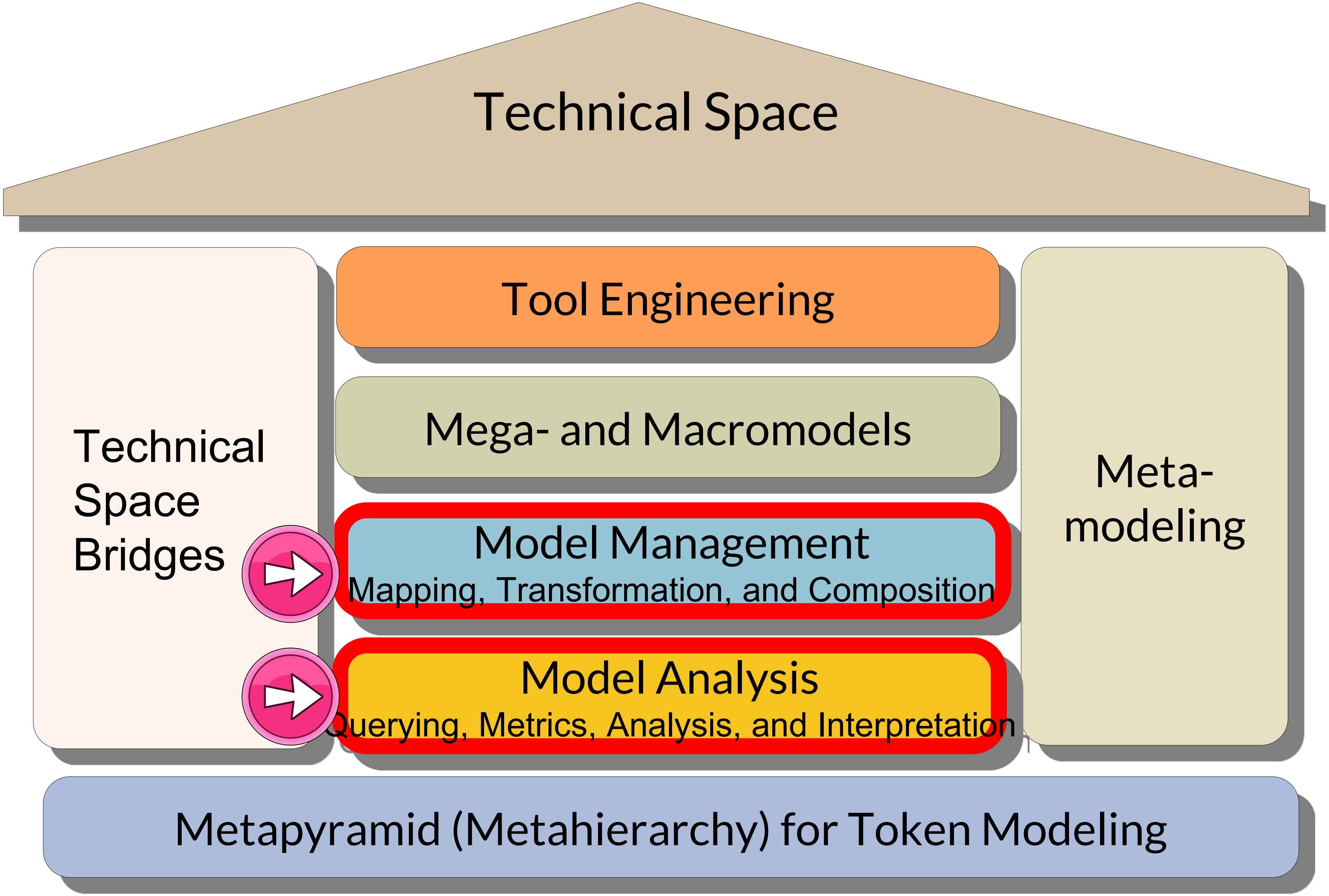
- [Hedin00] Hedin, Görel. 2000. Reference Attributed Grammars. Informatica (Slovenia) 24, Nr. 3: 301–317.
- [Boyland05] Boyland, John T. 2005. Remote attribute grammars. Journal of the ACM 52, Nr. 4: 627–687.
- [Knuth68] Knuth, D. E. Semantics of context-free languages. Theory of Computing Systems 2, Nr. 2: 127–145.
- [Vogt+89] Vogt, Harald H, Doaitse Swierstra, und Matthijs F Kuiper. 1989. Higher Order Attribute Grammars. In PLDI '89, 131–145. ACM. --- For code generation and template expansion.
- [Ekman06] Ekman, Torbjörn. 2006. Extensible Compiler Construction. University of Lund.
- [HM03] Görel Hedin, Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. Science of Computer Programming 47 (2003), pp. 37 – 58
  - www.elsevier.com/locate/scico
  - https://pdf.sciencedirectassets.com/271600/1-s2.0-S0167642300X01001/1-s2.0-S0167642302001090/main.pdf
- [MP20] Uwe Meyer, Björn Pfarr. Patterns for Name Analysis and Type Analysis with JastAdd. Technical Report. https://arxiv.org/abs/2002.01842

- GTTSE 09 is a very nice volume, downloadable under https://link.springer.com/book/10.1007/978-3-642-18023-1
- [H11] Hedin G. (2011) An Introductory Tutorial on JastAdd Attribute Grammars. In: Fernandes J.M., Lämmel R., Visser J., Saraiva J. (eds) Generative and Transformational Techniques in Software Engineering III. GTTSE 2009. Lecture Notes in Computer Science, vol 6491. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-18023-1_4
  - https://link.springer.com/chapter/10.1007%2F978-3-642-18023-1_4
- [H09] Hedin, G.: Generating Language Tools with JastAdd. GTTSE '09.
  - https://www.semanticscholar.org/paper/Tutorial%3A-Generating-Language-Tools-with-JastAdd-Hedin/e6a937a0fdd2673b08ddfa8f03a5e9cb6fef2efc
  - see also www.jastemf.org

RAGs, Template Expansion, Invasive Composition

- [Bürger+10] Bürger, Christoff, Sven Karol, und Christian Wende. 2010. Applying attribute grammars for metamodel semantics. In Proceedings of the International Workshop on Formalization of Modeling Languages, 1:1–1:5. FML '10. New York, NY, USA: ACM.
- Sven Karol. Well-Formed and Scalable Invasive Software Composition. PhD thesis, Technische Universität Dresden, May 2015.
  - http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-170162
  - Demonstrator Tool SkAT https://bitbucket.org/svenkarol/skat/wiki/Home.
- [Bürger15] Christoff Bürger. Reference attribute grammar controlled graph rewriting: motivation and overview. In Richard F. Paige, Davide Di Ruscio, and Markus Völter, editors, Software Language Engineering (SLE), pages 89-100. ACM, 2015. http://dl.acm.org/citation.cfm?id=2814251

Q10: The House of a Technical Space



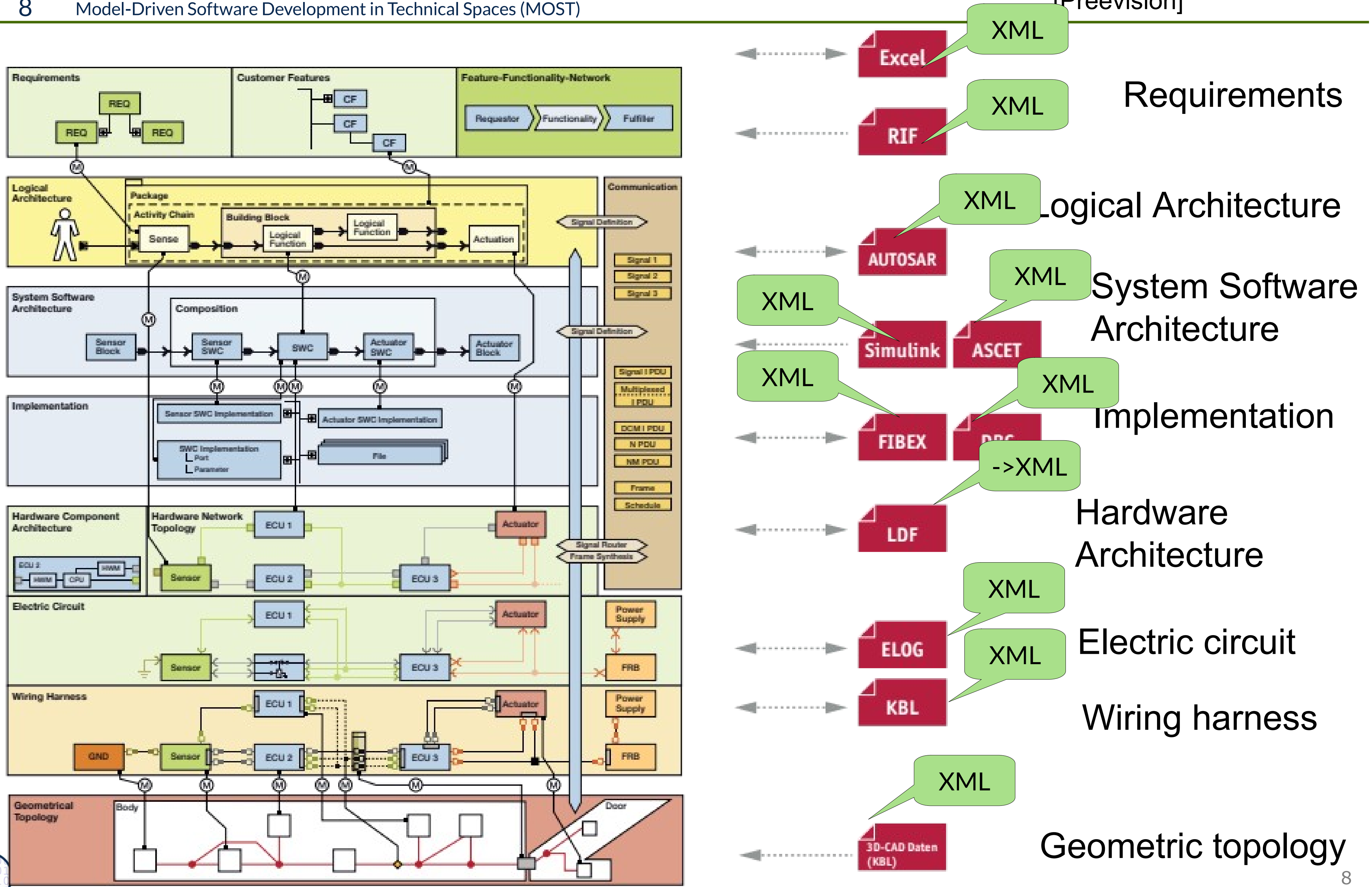
24.1 Reference Attribute Grammars for Interpreters and Analyzers on Syntax Link-Trees of Programs

- Interpretation and abstract interpretation on syntax link-trees

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Remember the Big Example: Car Design with PREEVision (Vector): Interoperability with XML Link Trees

[Preevision]



Link-Tree-Ware

- Main space for exchange between
  - Standalone, persistent tools
  - in Enterprise Architectures
  - in different technical spaces

Approach	Schema Language	
XML	XSD, RelaxNG, DTD	Node types
Ecore	EMOF	Node types
JSON	JSON schema	Nested dictionary (nodeless)
YAML, TOML		
RDF	RDFS	Triples in XML syntax

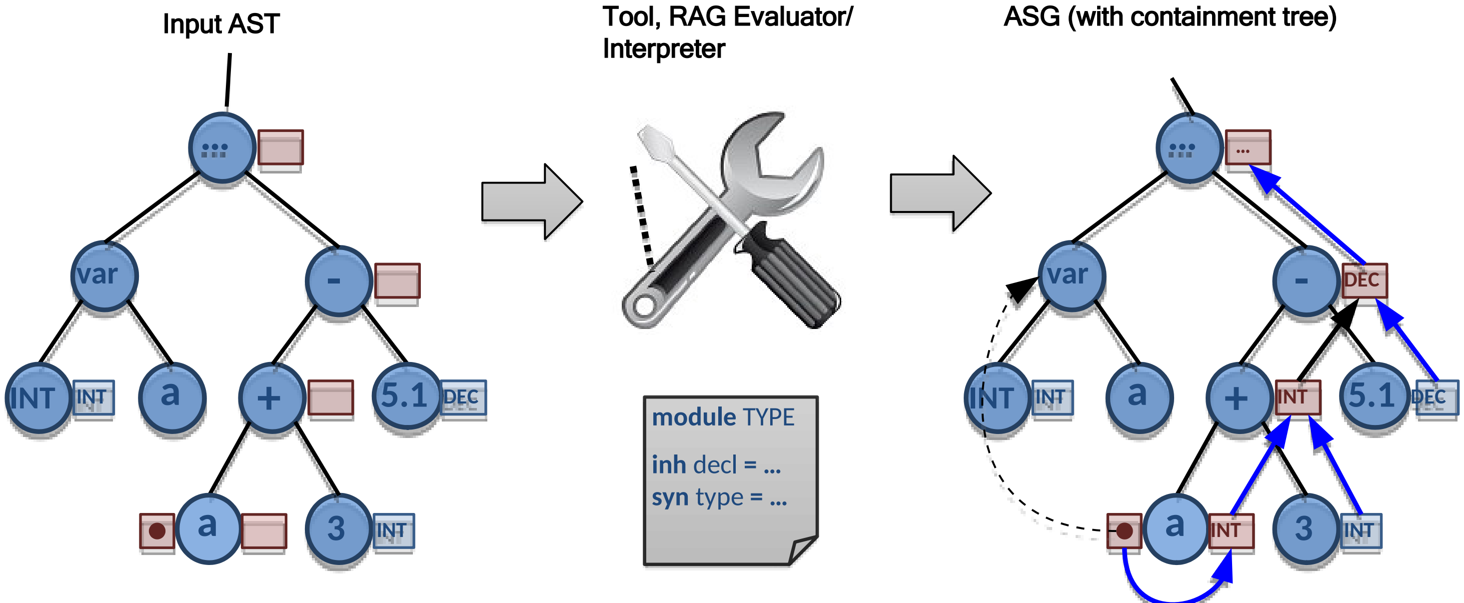
Rep.: Attribute Grammars (AG)

- An **attribute grammar** describes an interpreter on a syntax tree (a hierarchical program representation) computing an attribution from input to output values
  - The syntax tree is described by an RTG (or DTD, XSD) or context-free grammar (e.g., in EBNF)
  - The nodes of the program in the syntax tree are augmented with values, **attributes**. The resulting data structure is called **attributed syntax tree (AST)**
    - Graph representations are not possible in pure Ags
  - There is a set of **attribution rules (attribute equations, stencils)** defining interpretation functions on the syntax tree
  - Usually, the rules are interpreted with recursion along the attributed syntax tree
    - Rules **cover** the tree, i.e., every attribute has a computing function
  - Attribution rules do not rewrite, but compute attributions (stencils)
- *An attribute grammar describes an abstract interpreter*, if the values are from an abstract domain (e.g., from a type system, interval ranges, etc.)
  - Then, the set of **attribution rules (attribute equations)** define abstract interpretation functions on the syntax tree
- Because the underlying program representation is hierarchic, often
  - AG-based interpreters can be proven to terminate
  - can be compiled to code, instead of interpreted (pretty fast)

AG-based abstract interpreters can analyze syntax trees by abstract interpretation (deep analysis)

What is a Reference Attribute Grammar (RAG)?

- A ***reference attribute grammar*** can analyze link trees with attributions
  - **Attributions can compute „static semantics", „symbolic semantics", „collection semantics", or „abstract semantics" over syntax trees** [Knuth68]
  - Basis: (context-free) grammars + attributes + attribution (semantic) functions
- **Attribute types** and their **corresponding attributions:**
  - **Inherited attributes** (inh): Top-down value dataflow/computation (IN-parameters)
  - **Synthesized attributes** (syn): Bottom-up value dataflow/computation (OUT)
  - **Collection attributes** (coll): Collect values freely distributed over the AST
  - **Reference attributes**: Compute references (links) to exisiting nodes in the AST
- **Tool:** www.jastadd.org



Reference Attribute Grammars (RAG) Work on Link Trees

- A ***reference attribute grammar (RAG)*** describes an interpreter on a **syntax link-tree** with references to other branches (an overlay graph)
  - The syntax tree is described by an RTG (or DTD, XSD) or context-free grammar (e.g., in EBNF)
  - The *references* are described separately (e.g., links in XSD, JSON, EMF)
    - Overlay-graph representations *are* possible (***attributed link tree, ALT***)
  - The nodes of the program in the syntax tree are augmented with values, **attributes**
  - There is a set of **attribution rules (attribute equations)** which define interpretation functions on the syntax tree
  - Usually, the rules are interpreted with recursion along the syntax tree *plus* side recursions along the references
- *A reference attribute grammar describes an abstract interpreter*, if the values are from an abstract domain (e.g., from a type system, interval ranges, etc.)
  - Then, the set of **attribution rules (attribute equations)** define abstract interpretation functions on the syntax tree

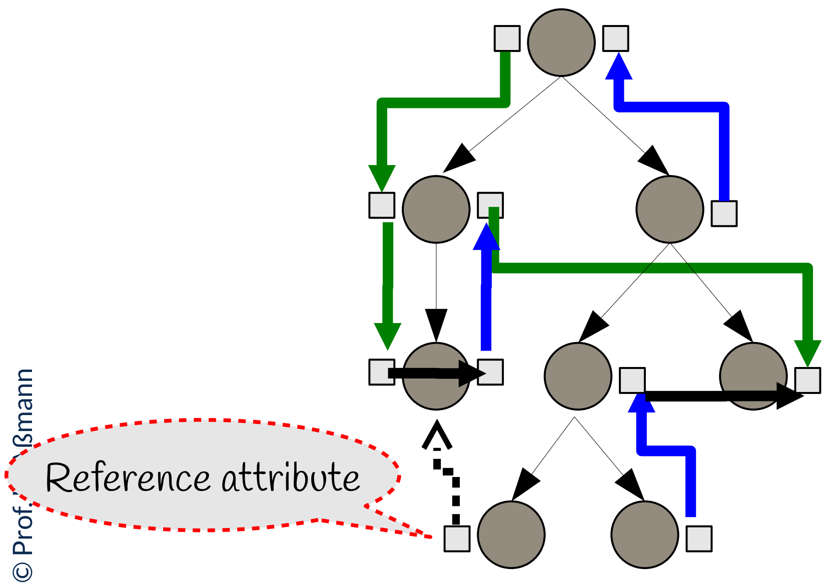
RAG-based abstract interpreters can analyse, interpret, and abstractly interpret models (deep analysis)

Link Tree Matching, Querying, Rewriting and RAG

- A RAG is defined to **cover** the tree with attributions
  - Matching, querying, rewriting does not need to cover the tree
- A RAG does **deep analysis**, i.e., specifies a computation of the *value flow* on the tree by interpreting expressions and assignments
  - By a global dependency graph between attributes and their attributions
  - Building dependency graphs between use and definitions of names

Kinds of Attributes and Attribute Dependencies

- Def.: A ***stencil (covering attribution)*** is an attribution that defines an attribute for *all* nodes in the tree
  - Not all attributions are stencils
- Stencils define data-flow, and corresponding data-dependencies between attributes of nodes (*attribute dependencies*)
- All attribute dependencies make up the ***attribute-dependency graph*** describing the value-flow in **deep analysis**



Local attributions for deep analysis by:

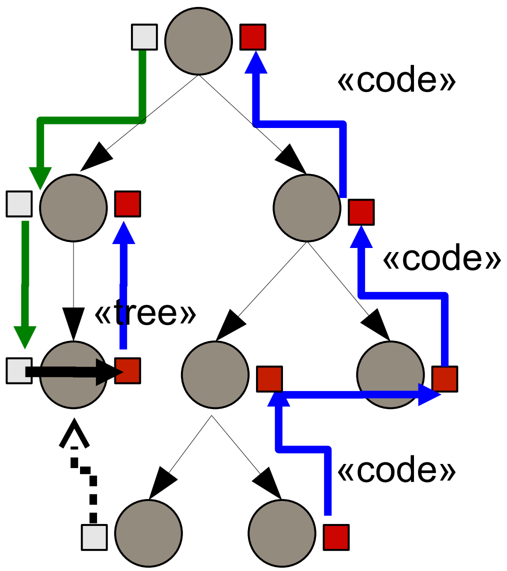
- **Inherited attributes** (inh, green): Top-down value dataflow/computation
- **Inherited attribution:** Stencil inherited from an ancester node, but applied locally
- **Synthesized attributes** (syn, blue): Bottom-up value dataflow/computation
- **Synthesized attribution:** Stencil inherited from an descendant node, but applied locally
- **Collection attributes** (coll): Collect values freely distributed over the AST

Non-local attributions for deep analysis by:

- **Reference attributes** (dashed): Compute references to non-local nodes in the AST

Kinds of Attributes (2) – Tree Computing Attributions

- AG and RAG can be used to compute trees in attributions
- A ***higher-order tree-generation attribution*** computes a new tree, may be from templates
- A ***higher-order tree-generation attribute*** stores the result of an higher-order attribution



Code generation attributions use higher-order attributes**:**

- **Higher-order (tree-generation) attributes** (inh and syn, blue): type of attribute is
  - `Tree:` ASTs are composed
  - `Code:` Code-snippets are composed
- **Template-expansion attributes**: computes tree from templates
  - `Tree:` AST-templates are composed
  - `Code:` Code-templates are composed

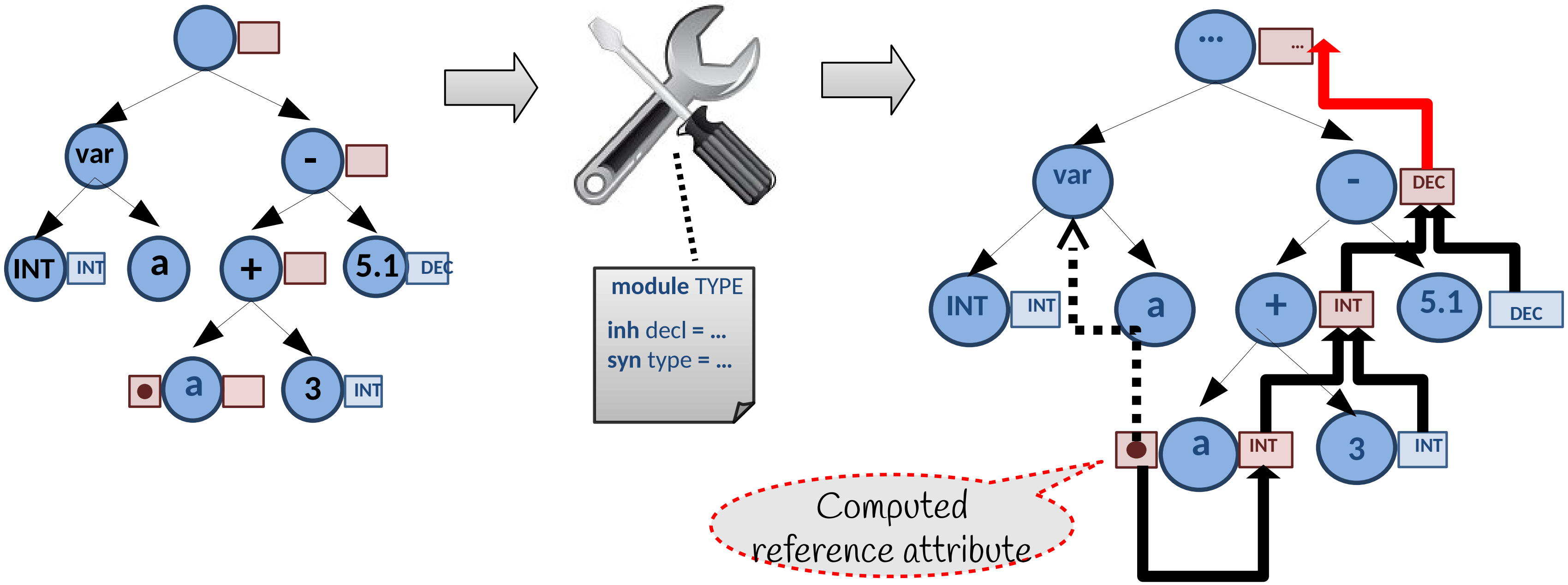
Basic Working Principle of RAG Tools

- Compute attribution functions
- Set references / links (dashed edge)

Input AST (from parser/ editor/transformer)

Tool: RAG Evaluator (generated or interpreted)

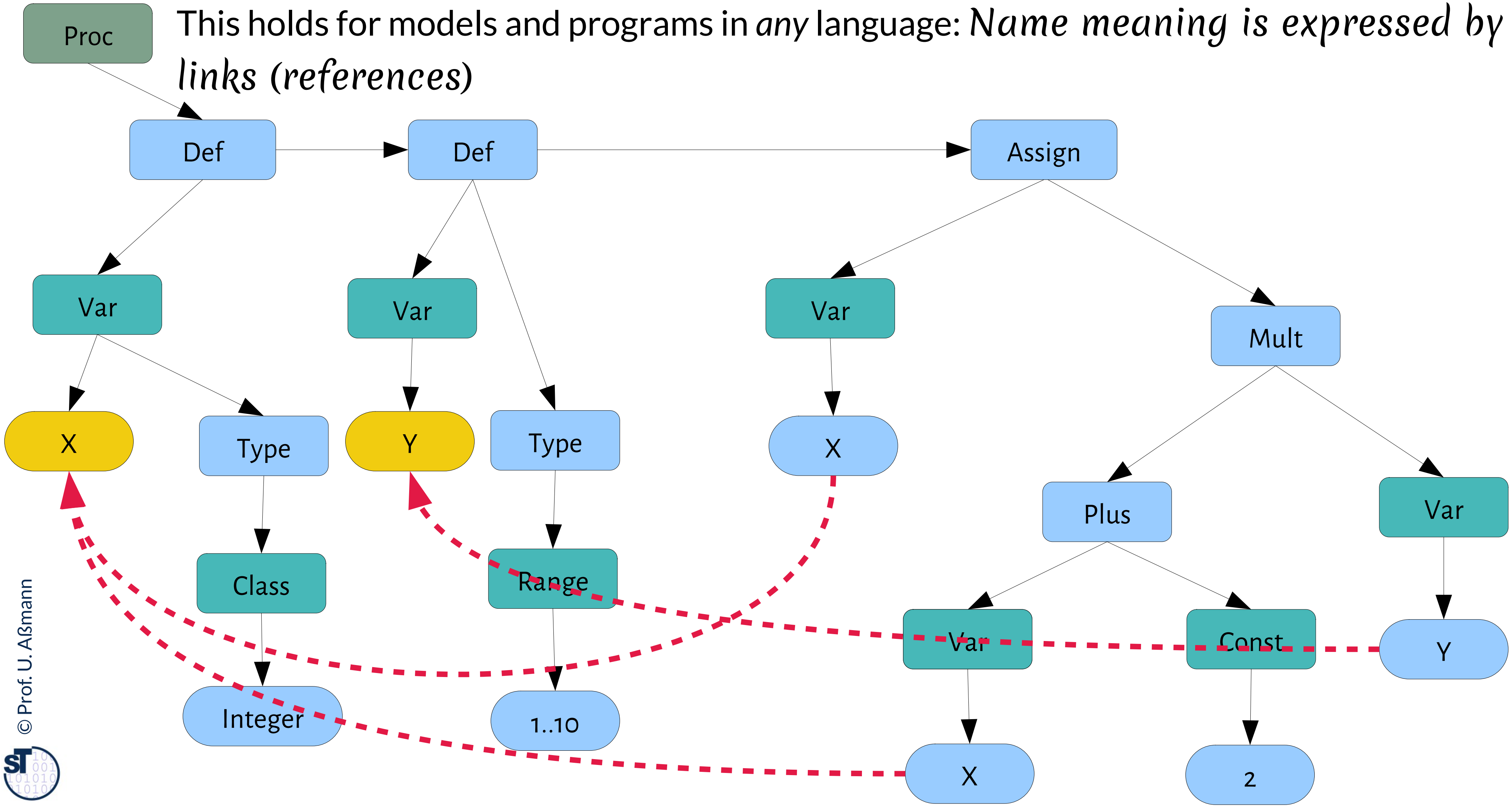
AST with overlay graph Attributed Link Tree (ALT)



Why Links? (1) Name Analysis (Name Resolution) in Programs and Models

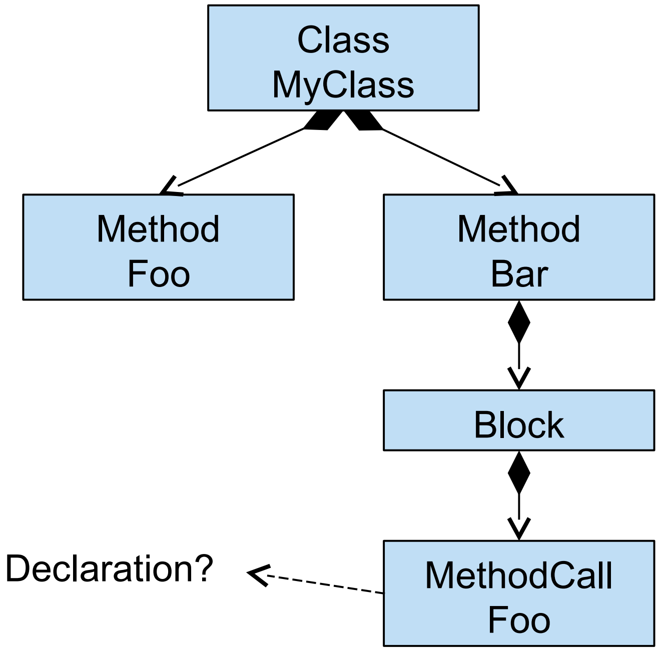
- ***Name analysis*** searches the right definition for a use of a variable and **materializes it as cross-tree link in an ALT**

This holds for models and programs in *any* language: *Name meaning is expressed by links (references)*



Why Links? (1) Name Analysis for Function Calls

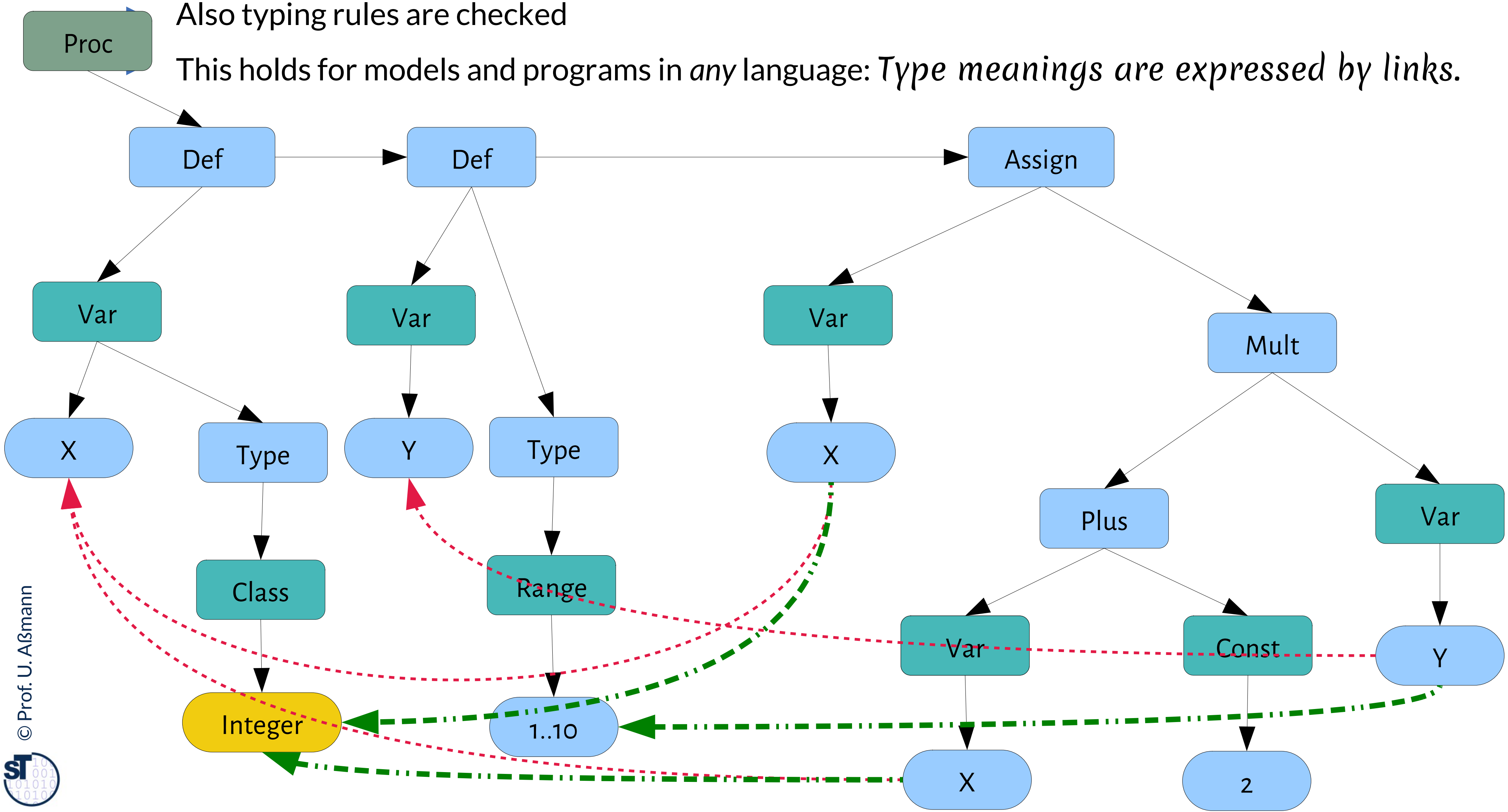
- ***Call-graph analysis*** searches the right definition for a call of a method and materializes it as cross-tree link (call graph)
- This holds for models and programs in *any* language: *Call relations are links.*



Practical problems:
Many possible name resolutions (Shadowing, overloading, several namespaces, namespace modifiers e.g. super, etc.)

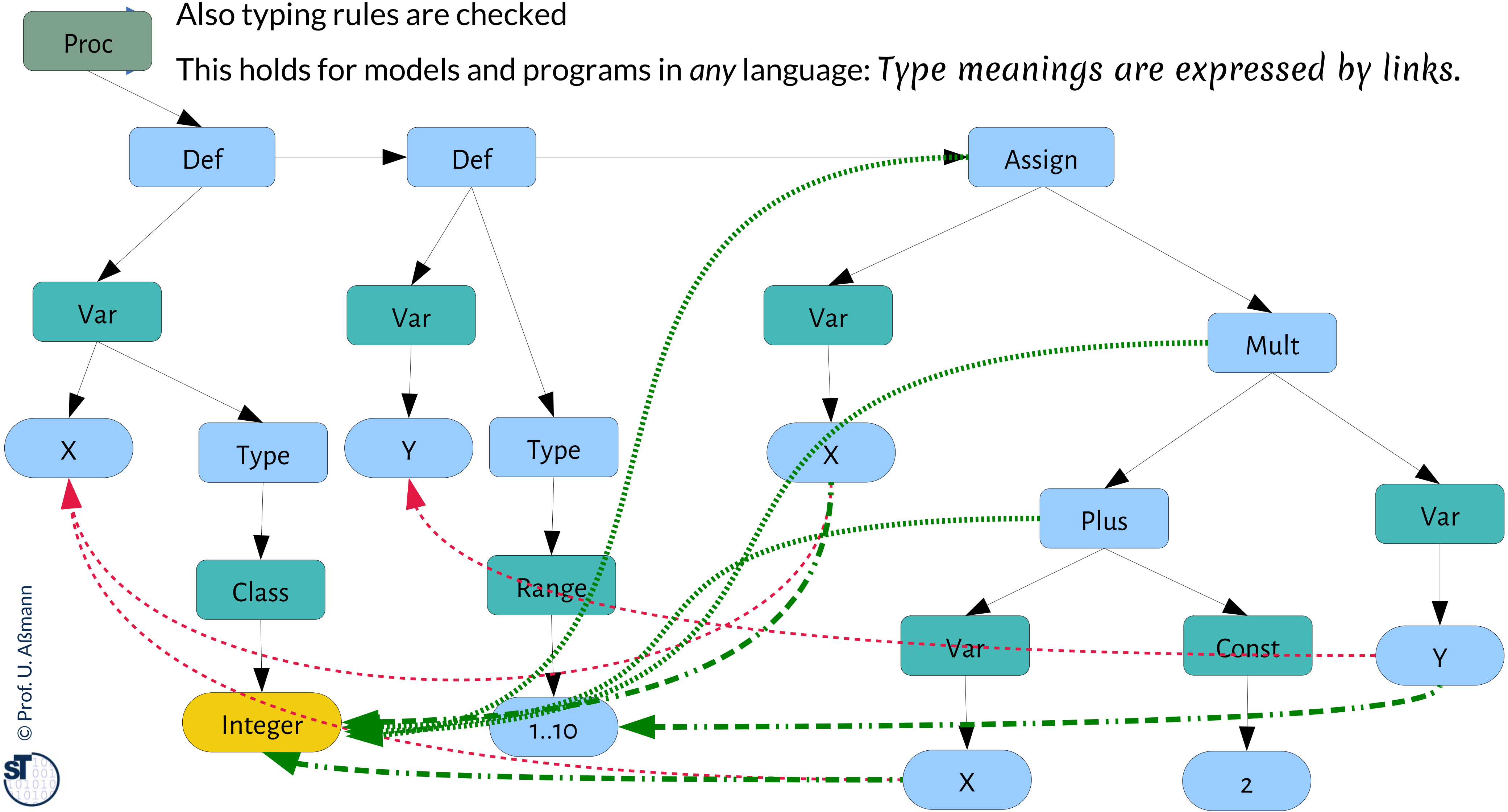
Why Links? (2) Type Analysis in Programs

- **Expressions in a program or model must be well-typed. Based on the meanings of names (their links), type analysis** searches the right types for a use of a variable *and for all expressions* and materializes them as cross-tree link (ALT)
- Also typing rules are checked
- This holds for models and programs in *any* language: *Type meanings are expressed by links.*



Why Links? (2) Type Analysis in Programs

- **Expressions in a program or model must be well-typed. Based on the meanings of names (their links), type analysis** searches the right types for a use of a variable *and for all expressions* and materializes them as cross-tree link (ALT)
- Also typing rules are checked
- This holds for models and programs in *any* language: *Type meanings are expressed by links.*



Deep Analysis of ASTs (Tree-Like Models)

- Deep analysis of ASTs *interprets the expressions and assignments in the AST*
  - producing ALTs, enriching the ASTs with links as results of the analysis
  - The links in the ALT provide the result of the analysis (reuse)

Links resulting from Static Semantic Analysis (Wellformedness Analysis):

- **Name analysis:** linking name references to definitions
- **Type analysis:** linking type references to definitions
- **Package analysis:** linking package names to definitions
- **Caller/Callee analysis**: linking callers to callees
- And many more

24.1.1 JastAdd Tool for Reference Attribute Grammars

- Data-driven programming on link trees shaped by RAGs
  - For link-treeware: EMF, JSON, XML, etc.

http://jastadd.org/web/documentation/concept-overview.php

The JastAdd RAG System
www.jastadd.org

JastAdd is an Object-oriented RAG evaluator generator

- Generated Java evaluators are demand-driven
- Handles combination of semantics, evaluation order and tree traversal
- Simple rewrite sublanguage
- Template expansion with higher-order synthesized attributes

Two specification languages (for AST nodes and attribution)

- For each AST node type a Java class is generated
- Access methods for child and terminal nodes are generated
- Each attribute represented by a method of a Java Class
- For each attribute equation a method implementation is generated
- **The generated class hierarchy is the attribute evaluator.**

JastAdd uses the parser generator Beaver for generating parsers

Example of SIPLE Programs

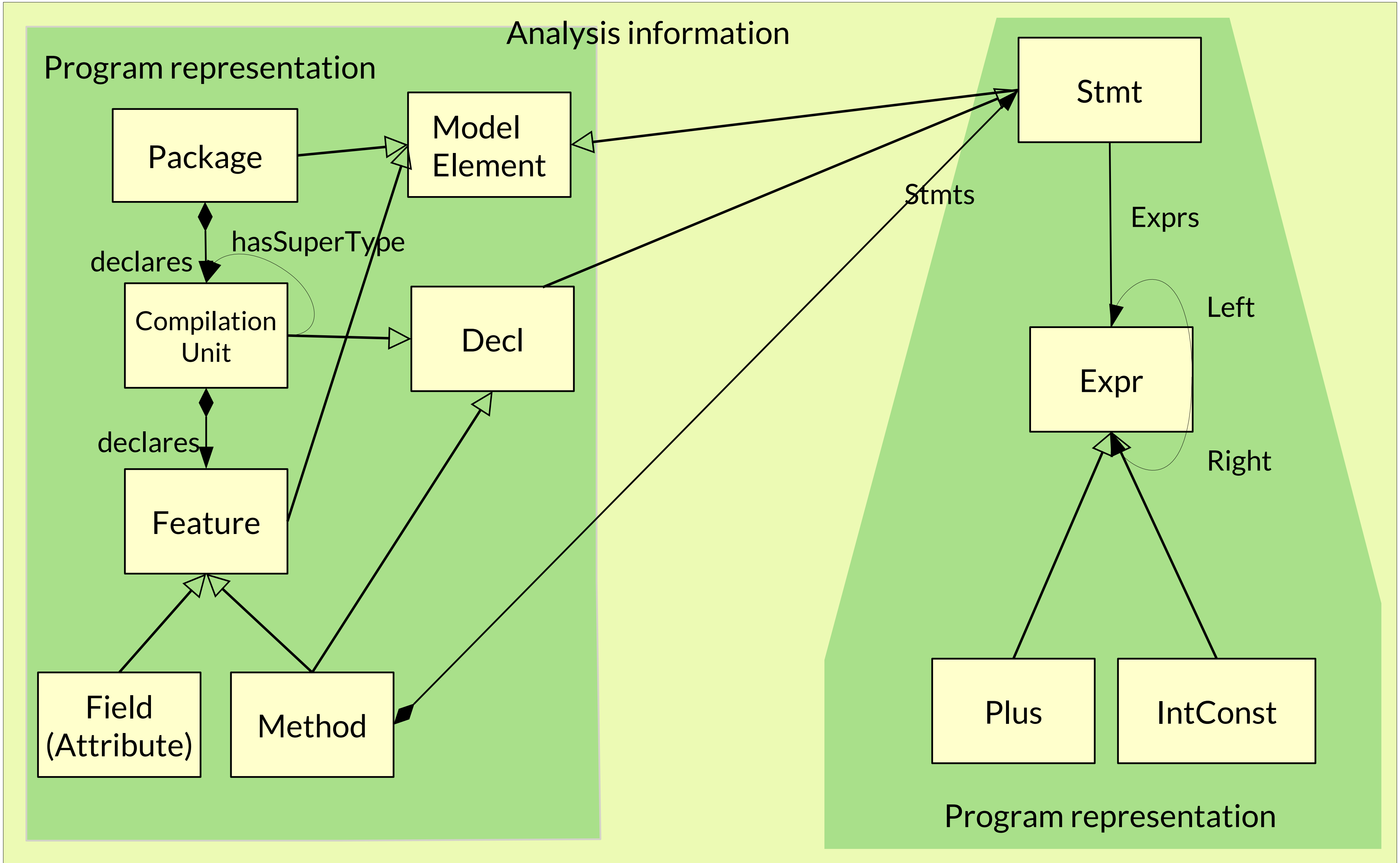
Example Language SiPLE, a Simple Prog. Lang. Beaver/JastAdd Grammar of SiPLE

[Bürger+10]
beaver.sourceforge.net/

https://bitbucket.org/jastemf/
https://bitbucket.org/jastemf/jastemf-plugins/src/4290860b492fcd10ac645b02eae64643cedf8192/jastemf-examples/siple/org.jastemf.siple/specifications/siple/syntax/parser.beaver?at=master&fileviewer=file-view-default


```
CompilationUnit = DeclarationList.decls
        {:  return new Symbol(new CompilationUnit(decls)); :}   // Motion syntax of Beaver
        ;
DeclarationList = Declaration.decl
        {: return new Symbol(new List<Declaration>().add(decl)); :}
        | DeclarationList.list Declaration.decl
        {: list.add(decl); return _symbol_list; :}
        ;
Declaration = VariableDeclaration.decl pSEMICOLON
        {: return _symbol_decl;    :}
        | ProcedureDeclaration.decl pSEMICOLON
        {: return _symbol_decl;   :}
        ;
VariableDeclaration = kVAR IDENTIFIER.id pCOLON Type.type
        {: return new Symbol(new VariableDeclaration(id, type));  :}
        ;
ProcedureDeclaration = kPROCEDURE IDENTIFIER.id pBRACKETOPENROUND ParameterList.paras pBRACKETCLOSEROUND pCOLON
Type.returnType Block.body
        {: return new Symbol(new ProcedureDeclaration(id, paras, returnType, body));         :}
        | kPROCEDURE IDENTIFIER.id pBRACKETOPENROUND pBRACKETCLOSEROUND pCOLON Type.returnType Block.body
        {: return new Symbol(new ProcedureDeclaration(id, new List<VariableDeclaration>(), returnType,
body));     :}
        | kPROCEDURE IDENTIFIER.id pBRACKETOPENROUND ParameterList.paras pBRACKETCLOSEROUND Block.body
        {: return new Symbol(new ProcedureDeclaration(id, paras, Type.Undefined, body));
        :}
        | kPROCEDURE IDENTIFIER.id pBRACKETOPENROUND pBRACKETCLOSEROUND Block.body
        {:return new Symbol(new ProcedureDeclaration(id,new List<VariableDeclaration>(),Type.Undefined, body)); :}
        ;
ParameterList = VariableDeclaration.decl
        {: return new Symbol(new List<Declaration>().add(decl)); :}
        | ParameterList.list pCOMMA VariableDeclaration.decl
        {: list.add(decl); return _symbol_list; :}
        ;
```

A Simple Model (Schema) of an OOPL in EMOF



```
// AST specification example:

abstract Stmt;          // Abstract nonterminals are like abstract classes
                        // in a metamodel
If:Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];  // inheritance :

abstract Decl:Stmt ::= <Name:String>;  // Attribute definition
ProcDecl:Decl ::= Para:VarDecl* Body:Block; // Containment links (kids)
VarDecl:Decl ::= <Type>;


// Attribution example in JastAdd:
syn Type Expr.Type();  // Type: Enumeration class of all types
eq BinExpr.Type() = …; // Default equation in a nonterminal
eq Equal:BinExpr = …;  // Subnonterminal refines equation
inh Block Stmt.CurrentBlock(); // Inherited attribute
// Assigning a reference attribute
eq Block.getStmt(int index).CurrentBlock() = this;
```

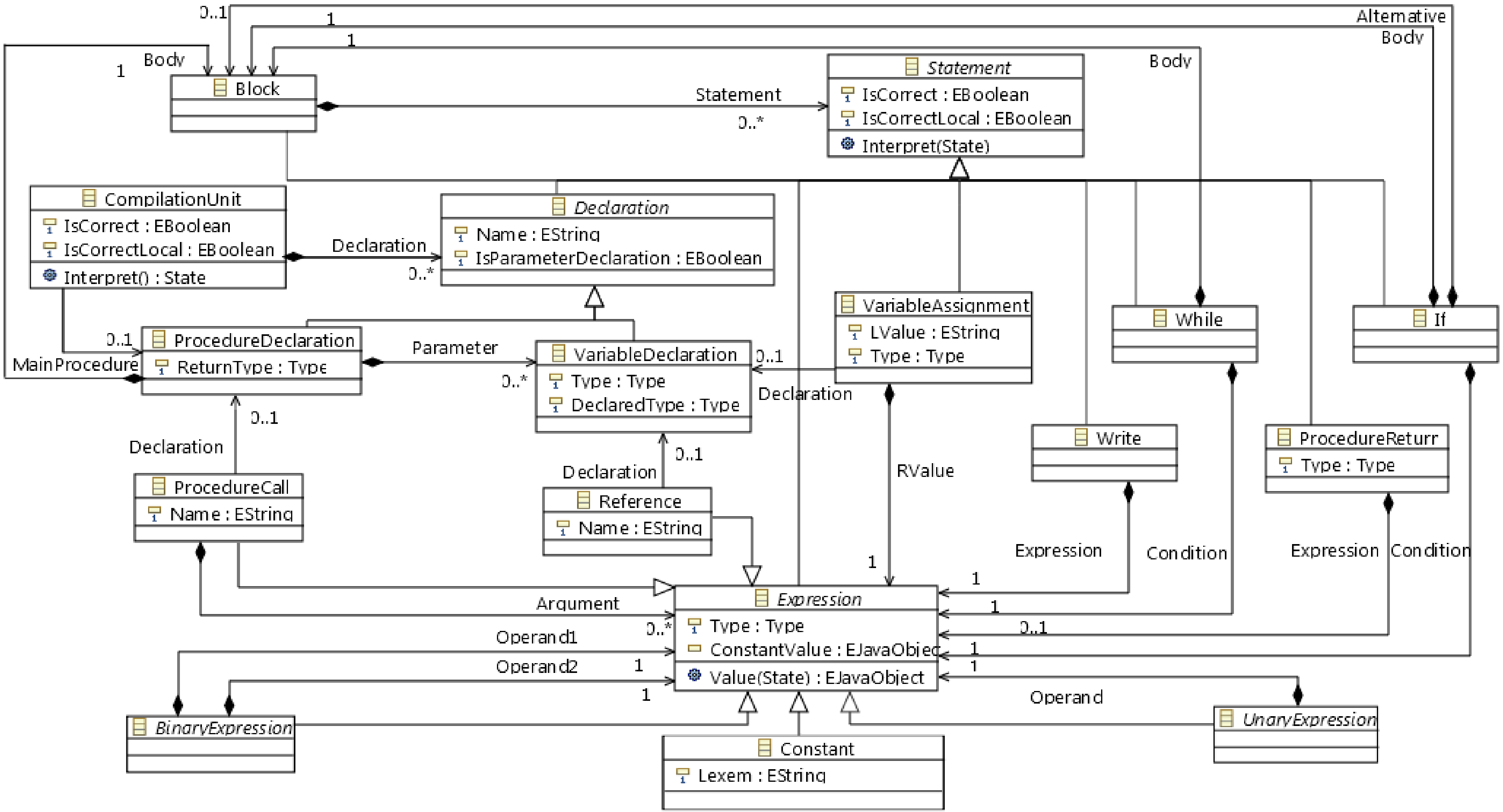
Example 1: SiPLE Grammar (in RTG Notation of JastAdd)

- SiPLE was discussed in [Bürger+10]

```
// Tree Grammar for SiPLE
CompilationUnit ::= Declaration*;

abstract Statement;

Block:Statement ::= Statement*;
If:Statement ::= Condition:Expression Body:Block
[Alternative:Block];
While:Statement ::= Condition:Expression
Body:Block;
VariableAssignment:Statement ::= <LValue:String>
RValue:Expression;
ProcedureReturn:Statement ::= [Expression];
Write:Statement ::= Expression;
Read:Statement ::= <LValue:String>;

abstract Declaration:Statement ::= <Name:String>;

ProcedureDeclaration:Declaration ::=
Parameter:VariableDeclaration*
<ReturnType:Type>
Body:Block;
VariableDeclaration:Declaration ::=
<DeclaredType:Type>;
```

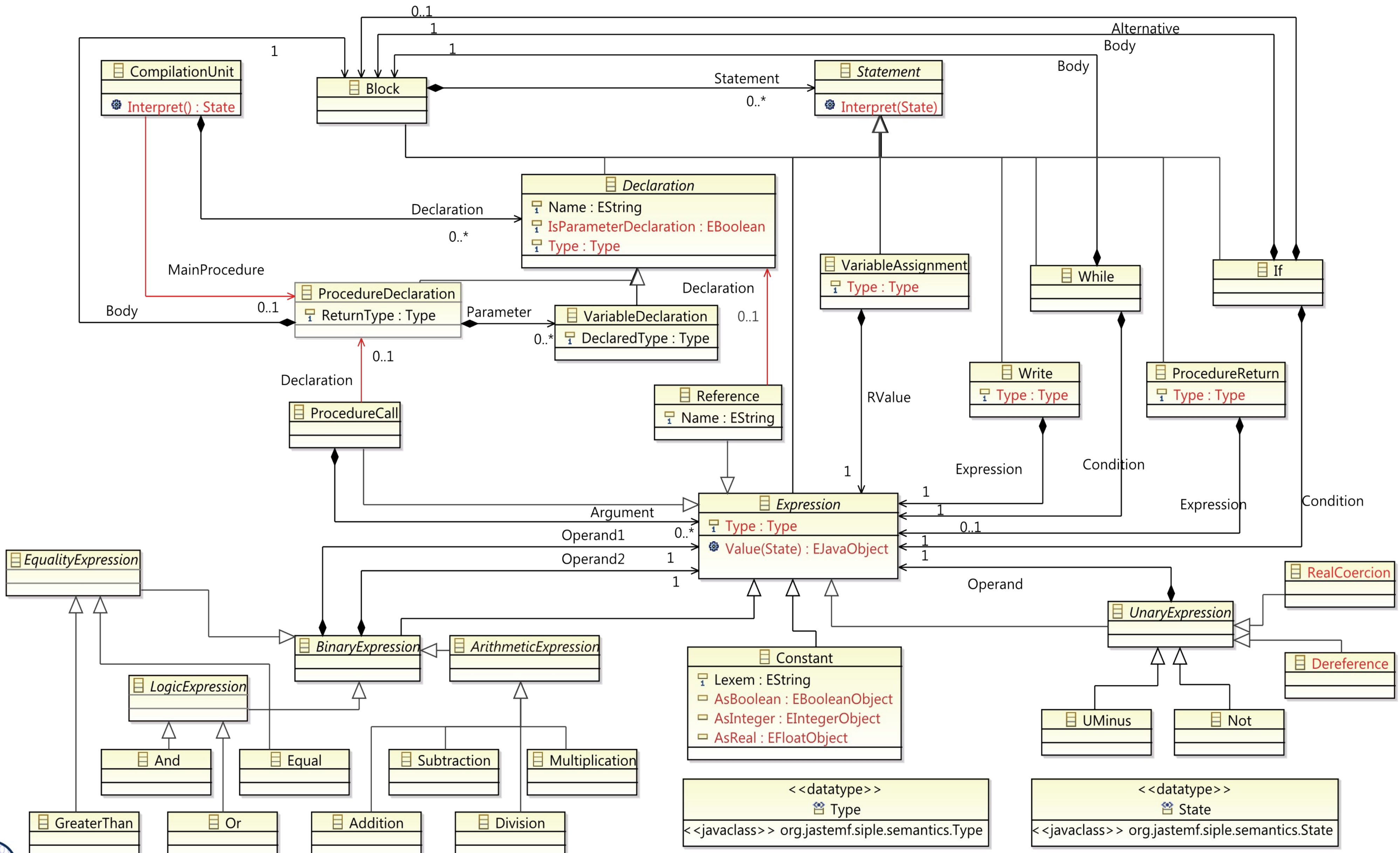
```
abstract Expression:Statement;

Constant:Expression ::= <Lexem:String>;
Reference:Expression ::= <Name:String>;
ProcedureCall:Expression ::= <Name:String>
Argument:Expression*;
NestedExpression:Expression ::= Expression;

abstract UnaryExpression:Expression ::=
Operand:Expression;

Not:UnaryExpression;
UMinus:UnaryExpression;

abstract BinaryExpression:Expression ::=
Operand1:Expression
Operand2:Expression;
```

Example 1: EMF Metamodel of SiPLE (Simple Programming Language)



Compare to SiPLE grammar.
Where is the spanning tree?

Example 1: SiPLE Programming Language EMF Metamodel

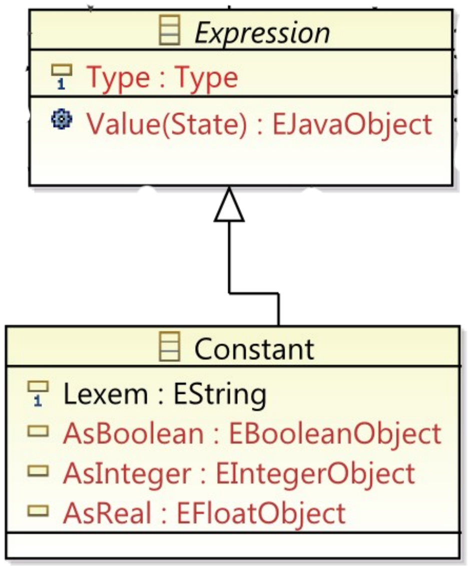


Example 1: SiPLE Types (Excerpt from Semantic Interface)

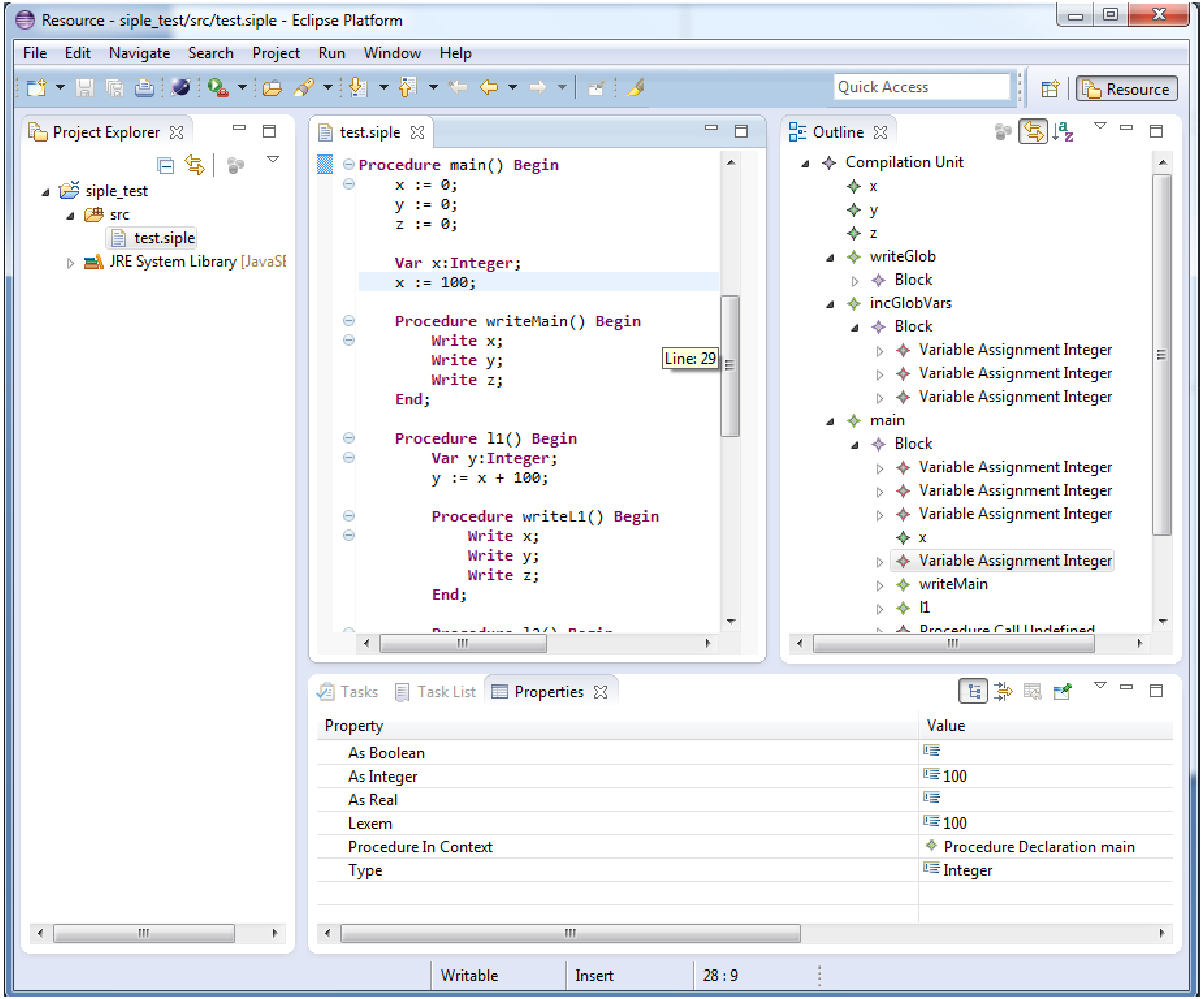

```
aspect TypeAnalysis {
          syn Type Declaration.Type();
          syn Type VariableAssignment.Type();
          syn Type ProcedureReturn.Type();
          syn Type Write.Type();
          syn Type Read.Type();
          syn Type Expression.Type();
}

aspect NameAnalysis {
          // Ordinary name space:
          inh LinkedList<Declaration> ASTNode.LookUp(String name);
          syn ProcedureDeclaration CompilationUnit.MainProcedure();
          syn Declaration Reference.Declaration();
}
```

Example 1: SiPLE Types (Excerpt from Definitions)


```
/** Expressions' Type **/

eq Constant.Type() {
        if (AsBoolean() != null)
                return Type.Boolean;
        if (AsReal() != null)
                return Type.Real;
        if (AsInteger() != null)
                return Type.Integer;
        return Type.ERROR_TYPE;
}
```



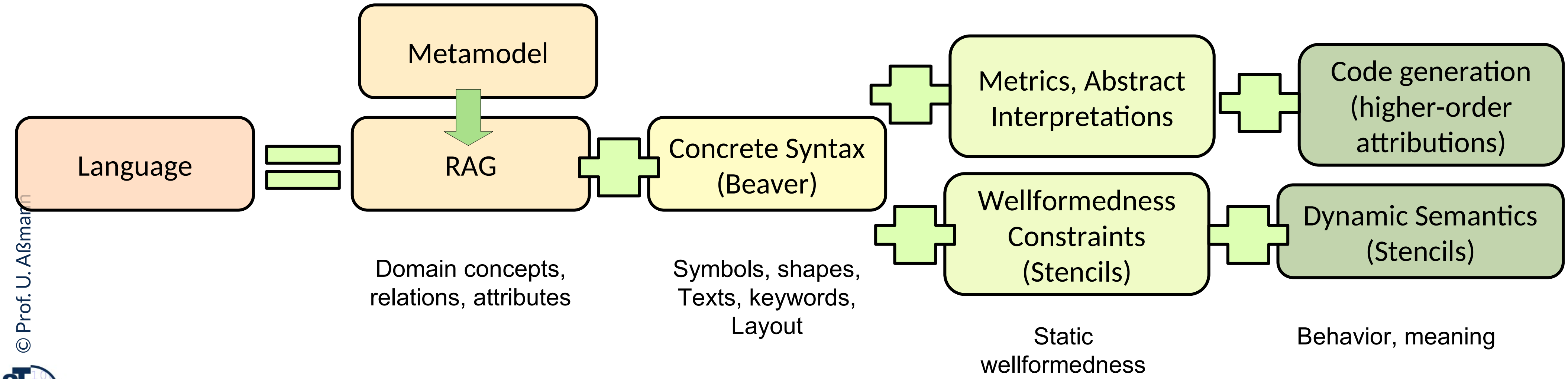
SiPLE Eclipse Editor



RAGs for Textual Language Interpretation and Analysis

Compiler-Frontends for Textual Languages can produced with JastAdd (RAG)

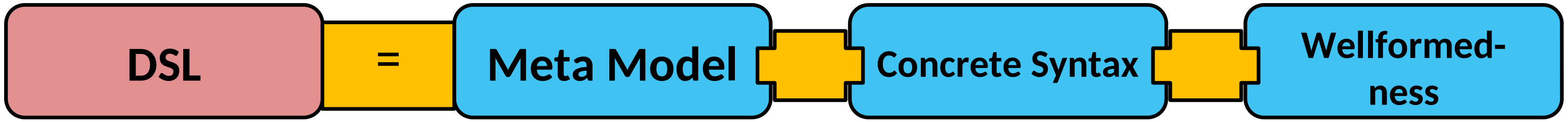
- After parsing, the RAG processes links for the pure tree
  - Completing the link tree with references to an ALT
  - Name analysis, type analysis, wellformedness constraints
- Metrics by attributions
- Abstract interpretations by attributions
- Template expansion for code generation
  - as well as Invasive composition (template extension)
- Program interpretation by attributions

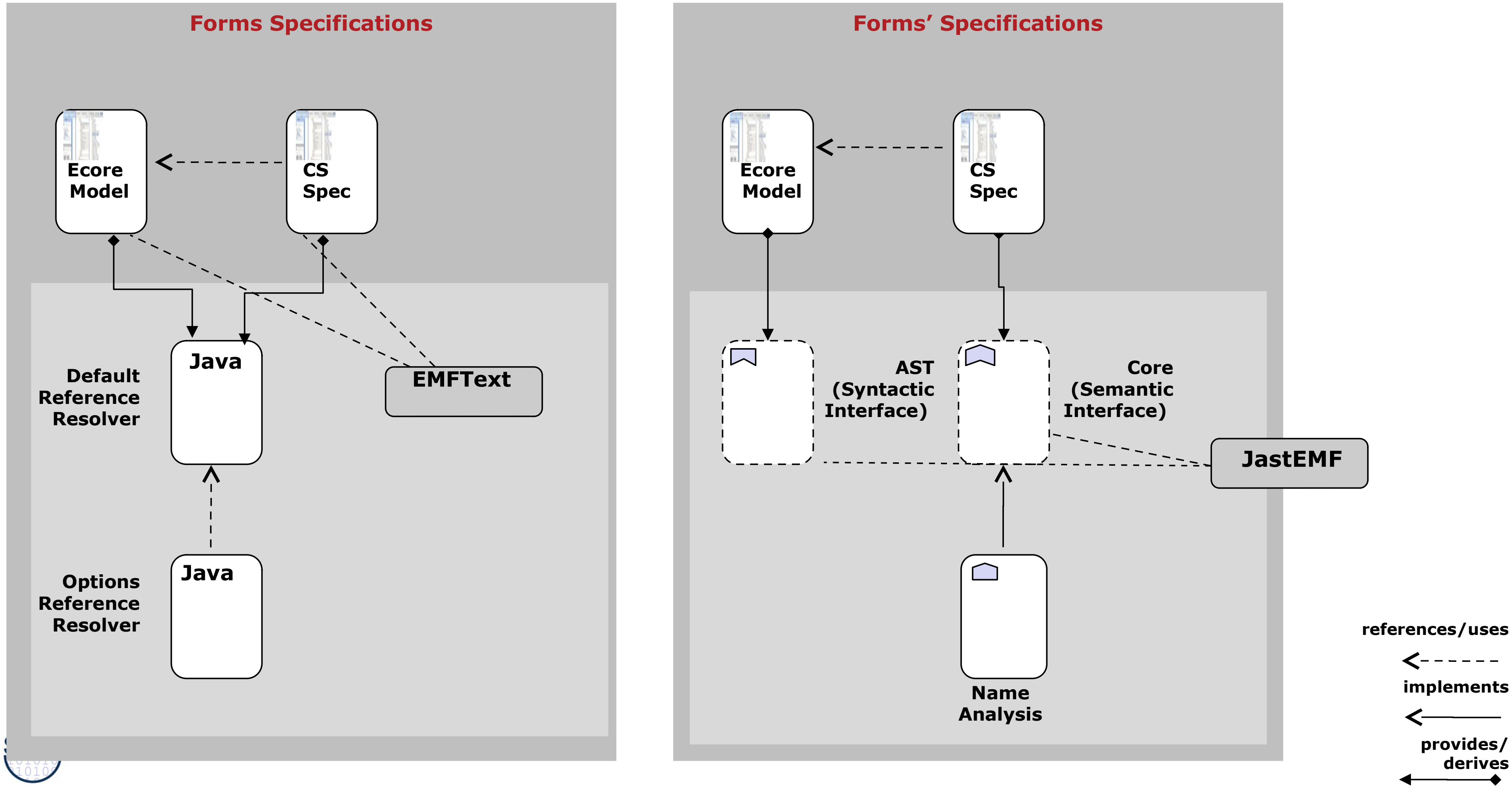


24.1.2 JastAdd for DSL

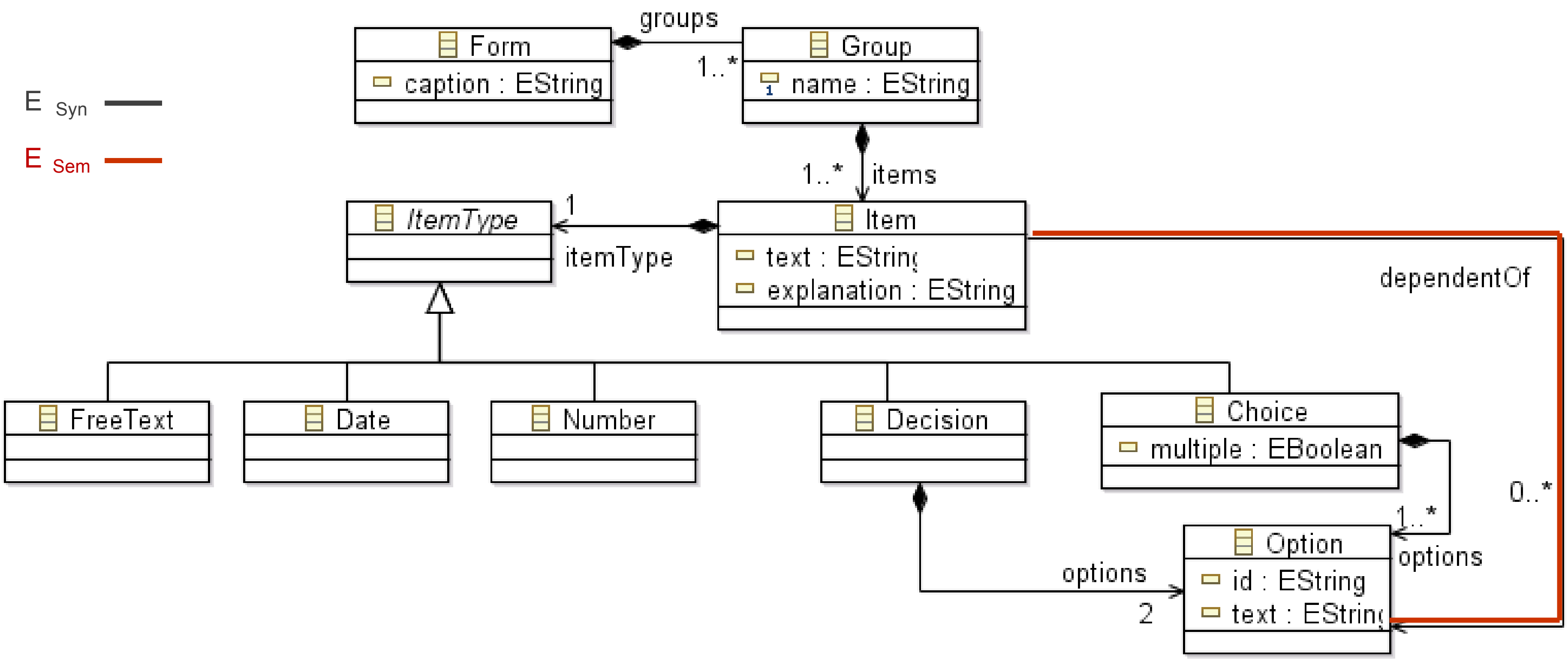
- Domain-Specific Languages

Example 2: Forms DSL





Example 2: Forms Metamodel



Example 2: Domain-Specific Language "Forms" as JastAdd Grammar

Terminal - EAttribute

ASTNode - EClass

```
Form ::= <caption:String> groups:Group*;

Item ::= <text:String> <explanation:String> <dependentOfName:String>
           itemType:ItemType;


abstract ItemType;
FreeText:ItemType;
Choice:ItemType ::= <multiple:boolean> options:Option*;
Date:ItemType;
Number:ItemType;
Decision:ItemType ::= options:Option*;

Option ::= <id:String> <text:String>;

Group ::= <name:String> items:Item*;
```

Inheritance ':'

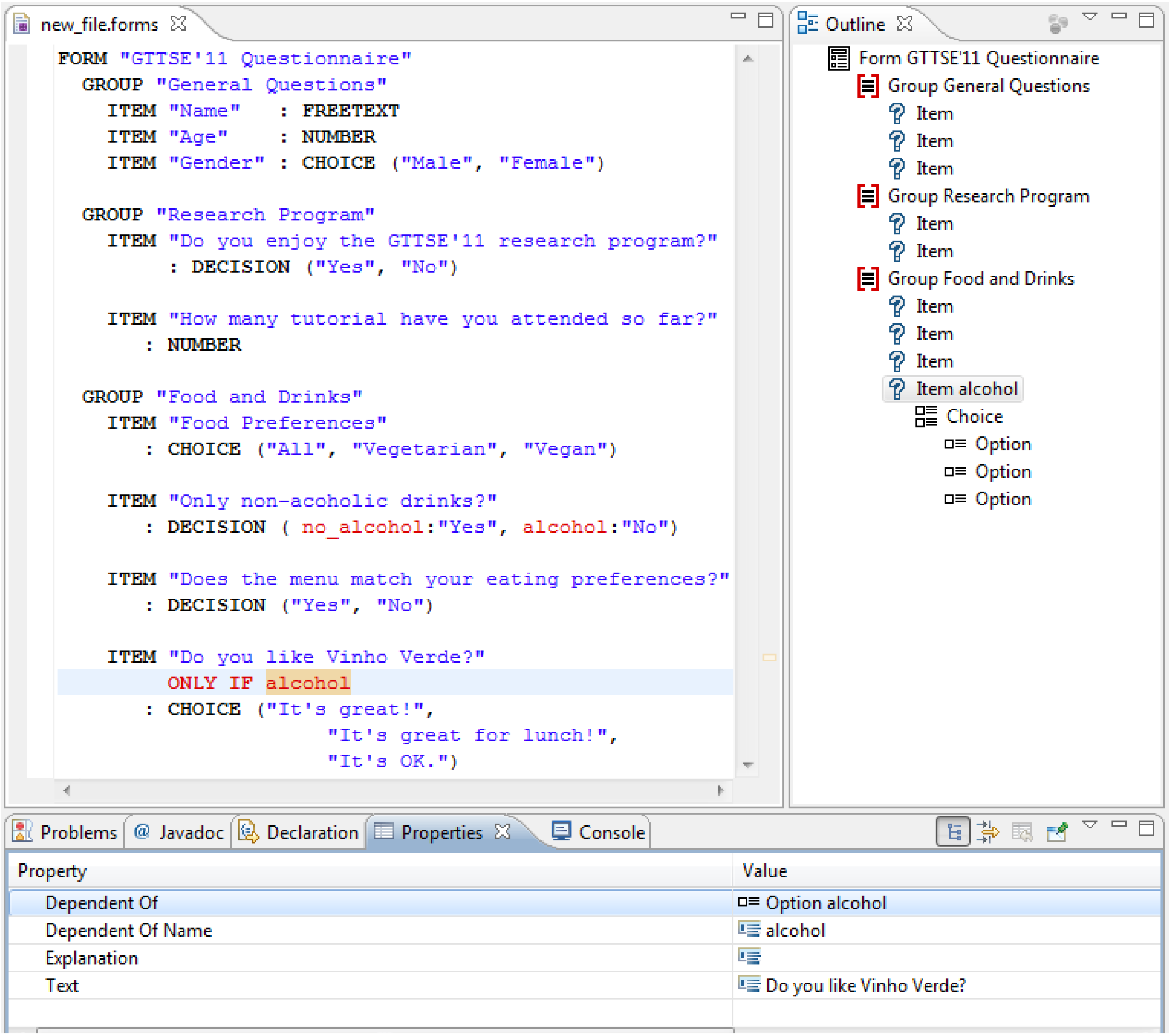
Nonterminal - EReference (Containment)

Example 2: Forms Attributes

- **Aspect modules** specify extensions of tree nodes, attributes, and attributions

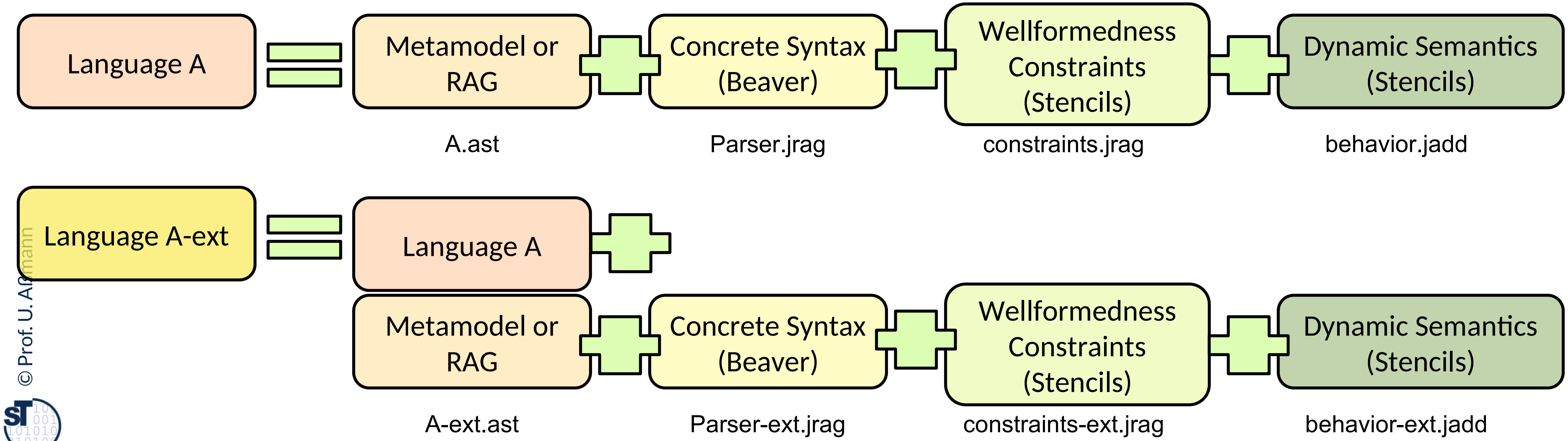
```
aspect NameAnalysis {
   inh Form ASTNode.form();
   syn EList Item.dependentOf();
   inh EList ASTNode.LookUpOption(String optionName);
   coll EList<Option> Form.Options() [new BasicEList()] with
   add;

   Option contributes this to Form.Options() for form();

   eq Form.getgroups(int index).form() = this;
   eq Item.dependentOf() = LookUpOption(getdependentOfName());

   eq Form.getgroups(int index).LookUpOption(String optionName){
           EList result = new BasicEList();
           for(Option option:Options()){
                   if(optionName.equals(option.getid()))
                           result.add(option);
           }
           return result;
   }
}
```

Forms Editor (Eclipse)



Extending a Base JastAdd Specification, e.g., for an Embedded DSL

- Extensions are simple, because arbitrary many definitions may be given in `.jrag` and `.jadd` files, which are *merged* by JastAdd
  - Merging works because attributions are functional programs without side effects
  - Merging doesn't work if programmers program side effects via Java
- Application: Base languages can easily be extended by extensions
  - ***Embedded DSL***: DSL embedded in a base language

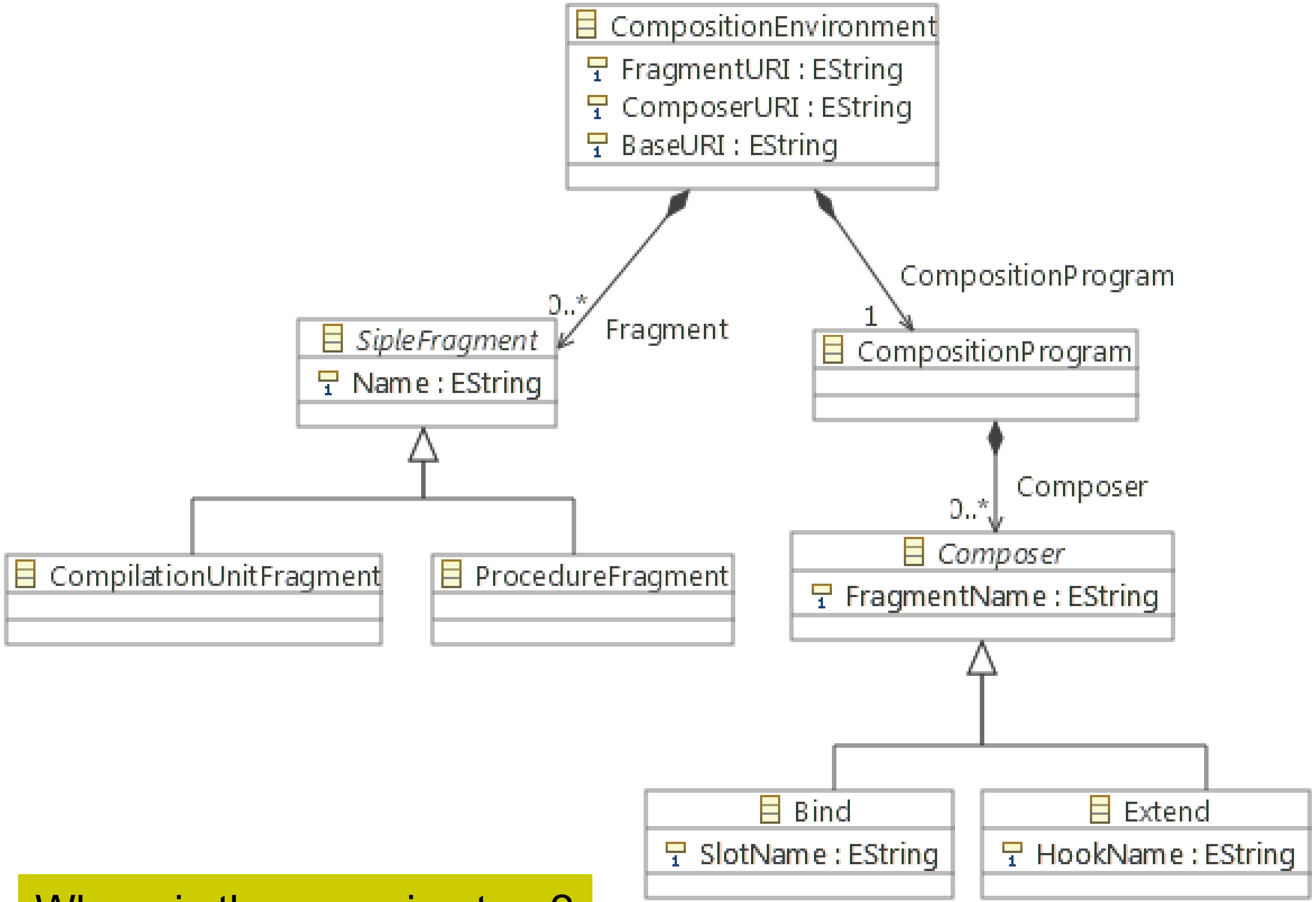


24.1.3 Extending Languages Made by JastAdd

- Domain-Specific Languages

Template-SiPLE, a Simple Extension of SiPLE EMF Metamodel (for the AST)

Template-SiPLE is a simple extension of SiPLE with templates („generic fragments“), template parameters and template composition operators (Bind, Extend a template parameter)



Where is the spanning tree?

Template-SiPLE Aspect Extension of SiPLE (in RTG Notation of JastAdd)

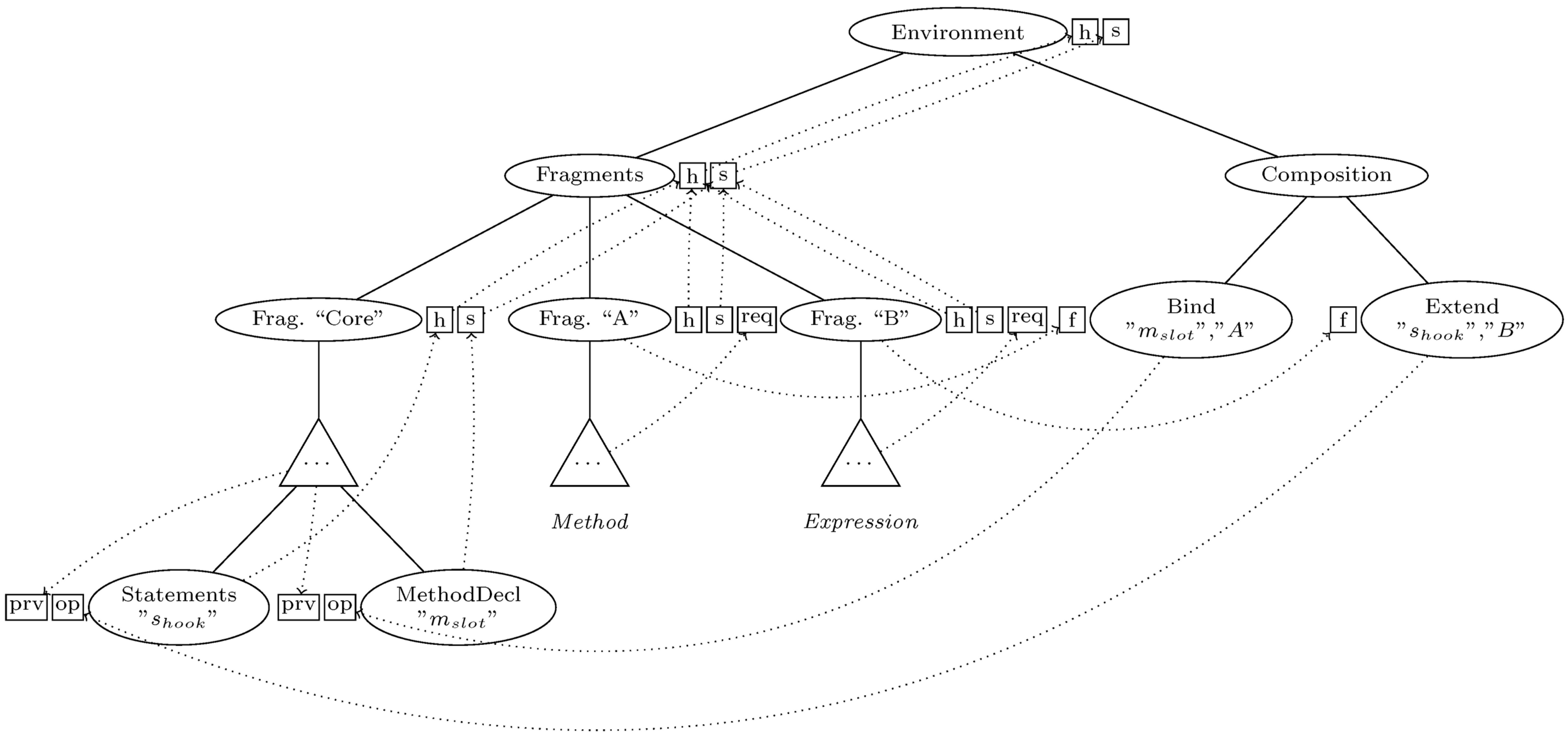
- JastAdd **aspect modules** specify *cross-cutting extensions* of tree nodes, attributes, and attributions [HM03]
- Template-SiPLE extension can be specified in an aspect module

```
// Tree Grammar Aspect for Template-SiPLE
aspect TemplateSiPLE {
 CompositionEnvironment  ::=
<FragmentURI:String> <BaseURI:String>
<ComposerURI:String>;

 abstract SipleFragment ::= <Name:String>;

 CompilationUnitFragment:SipleFragment;
 ProcedureFragment:SipleFragment;

 // Extension of SiPLE grammar for
 // composition in CompositionProgram
 /// Template Parameters in Declaration
 Declaration ::= <SlotName:String>;
 /// Extension Points in Declaration
 Declaration ::= <HookName:String>;
 /// Template Parameters in Expression
 Expression ::= <SlotName:String>;
 /// Extension Points in Expression
 Expression ::= <HookName:String>;
```

```
 abstract CompositionProgram;

 CompositionProgram ::= Composer*;


 abstract Composer::= <FragmentName:String>;

 Bind:Composer ::= <SlotName:String>;
 Extend:Composer ::= <HookName:String>;

}
```

See chapter “Invasive Composition” in CBSE course (SoSe)

RAG Analysing Template-SIPLE Composition Program



[Karol14]

24.2 Reference Attribute Grammars for Interpreters and Analyzers on Attributed Link-Trees of Models

- Interpretation and abstract interpretation on syntax link-trees with the tool JastEMF (1.0)
- https://bitbucket.org/jastemf/jastadd2-emf/src/master
- https://bitbucket.org/jastemf/jastemf-plugins/src/master/
- At the moment, we work on a JastEMF 2.0, in which JastAdd reads Ecore directly

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Claim (see EMFText):

Most metamodeling languages' metamodels separate model instances into

- A tree structure (AST) and
- A link-graph structure based on references between tree nodes (ALT, ASG)

Facts:

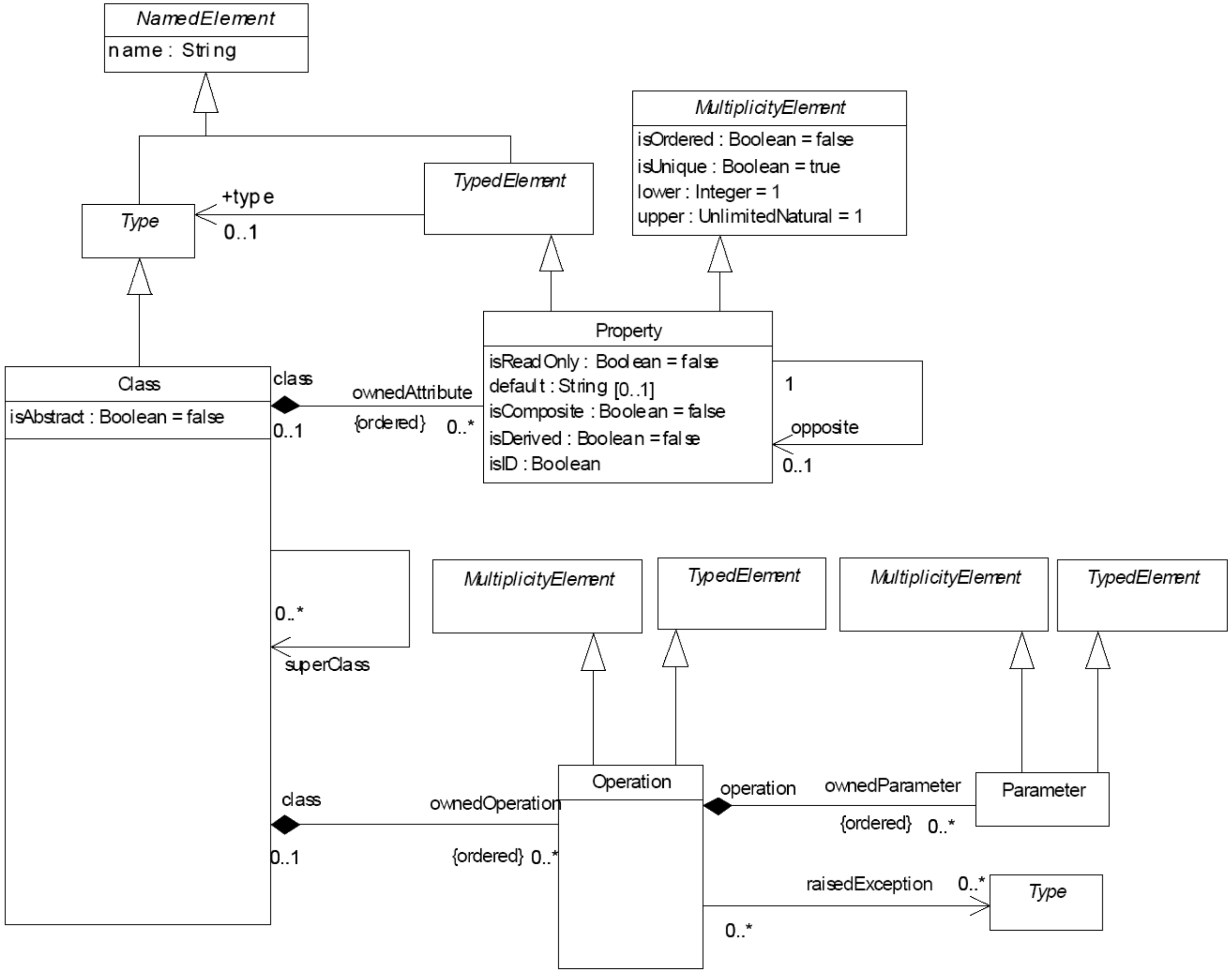
- Metamodeling standards often provide so called *metaclasses*, *containment references* and *non-derived properties* to model ASTs
- In language theory and compiler construction *context-free grammars (CFG)* and *regular tree grammars (RTG)* specify context-free structures (ASTs)
- Reference attribute grammars (RAGs) are a well-known concept to specify ALTs based on ASTs and to reason about ALTs
- EMFText resolvers can be written with RAGs

Since both approaches look so similar, why not combine them?

EMOF/Ecore Revisited: Link-Tree Structure and Semantics

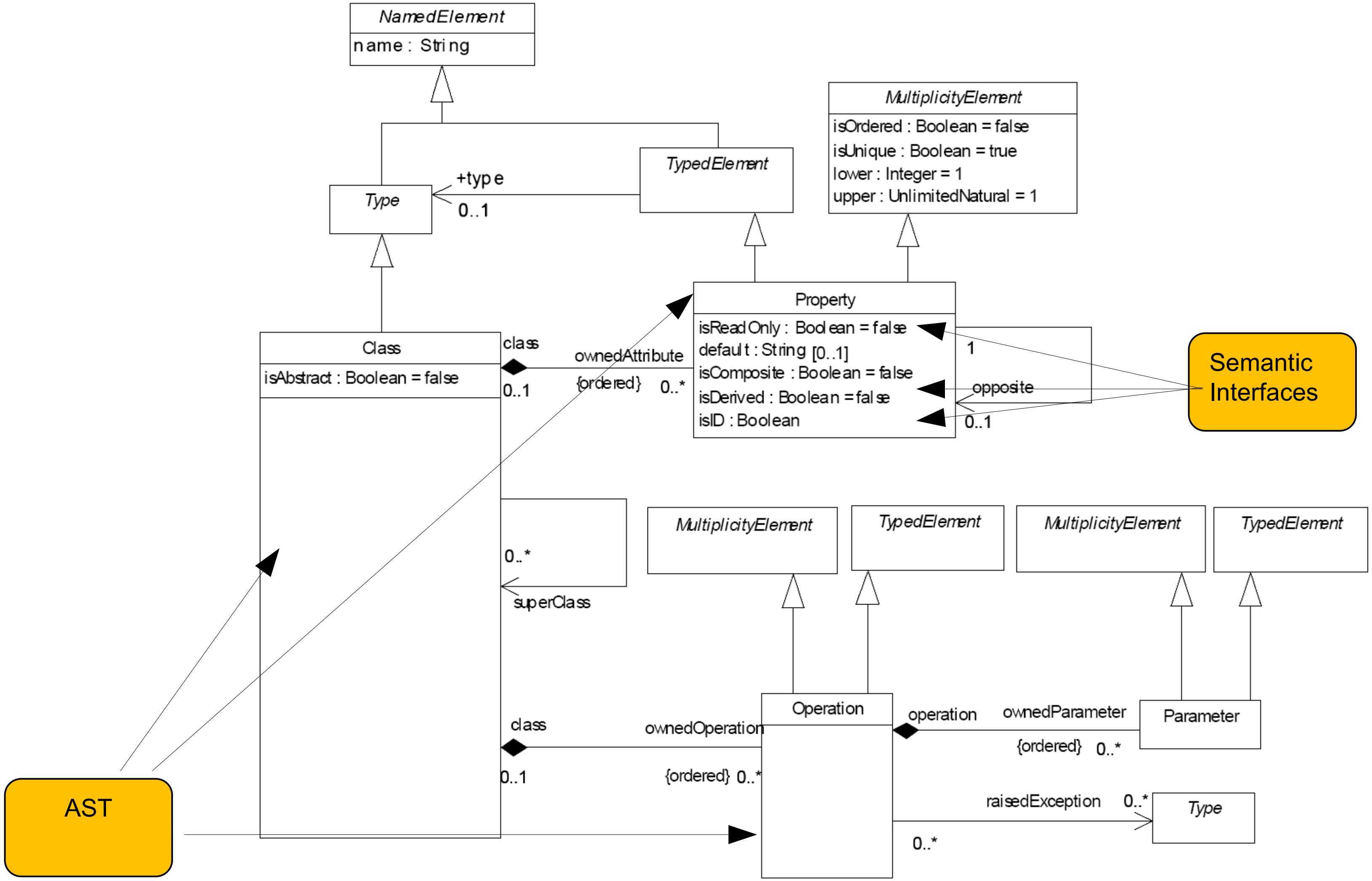
- Each model instance of an Ecore metamodel has a spanning tree of containment references
  - Its set of nodes are all metaclass instaces (Non-terminals) and non-derived properties (Terminals)
- Model instances' semantics are
  - Derived properties (ALT)
  - Non-containment references (ALT)
  - Operations
- Derived properties and non-containment references = Attributed Link Tree (ALT) on top of the spanning tree.

The EMOF Metamodel – What is Syntax, What is Static Semantics?



Where is the spanning tree?

The EMOF Metamodel – What is Syntax, What is Static Semantics?



The JastEMF Approach Requires a Ecore-JastAdd Concept Mapping

In summary: EMF and JastAdd generate a class hierarchy

- EMF generates:
  - Metamodel implementation (Repository + Framework/Editors etc.)
  - AST structure derived from aggregations
  - Accessor methods (Implementation for AST; Skeletons for semantics)
- JastAdd generates:
  - Evaluator implementation
  - Accessor methods for AST + Semantic implementation

EMF metamodel implementation (Repository)

+

JastAdd semantic methods working on the repository

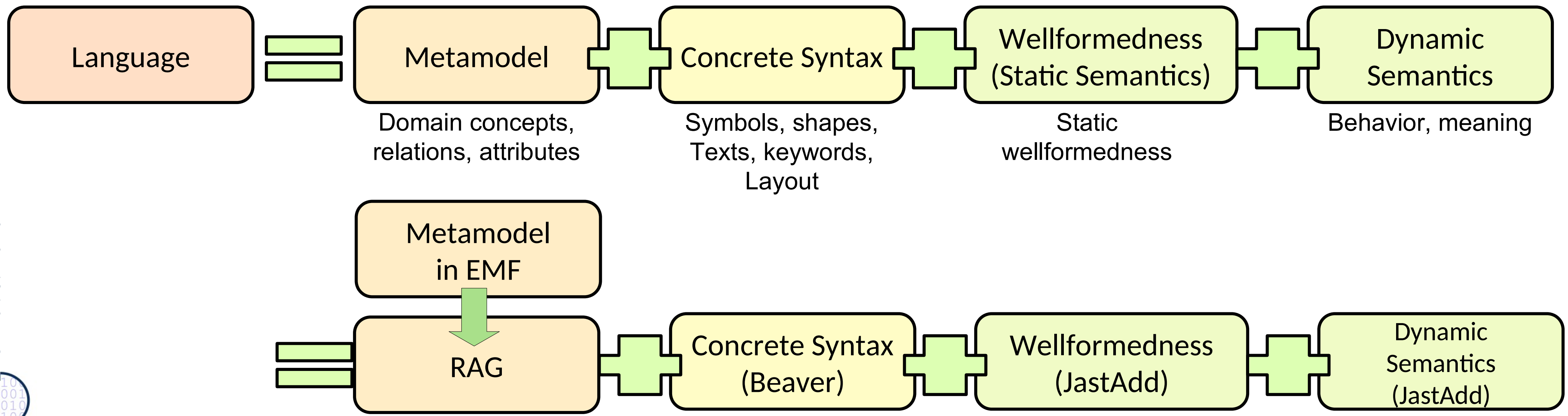
=

Semantic metamodel implementation

The JastEMF Approach Requires a Ecore-JastAdd Language Mapping (Concept Mapping)

Idea: EMF metamodel implementation (Repository) + JastAdd semantic methods working on the repository = semantic metamodel implementation

- For every *derived property*: JastAdd attribute of equal name and type
- For every *non-containment reference*: JastAdd reference attribute of equal name and type
- For *side-effect free operations*: JastAdd attribute of equal signature
- *Metamodel AST* (Metaclasses; non-derived properties; containment references): JastAdd AST

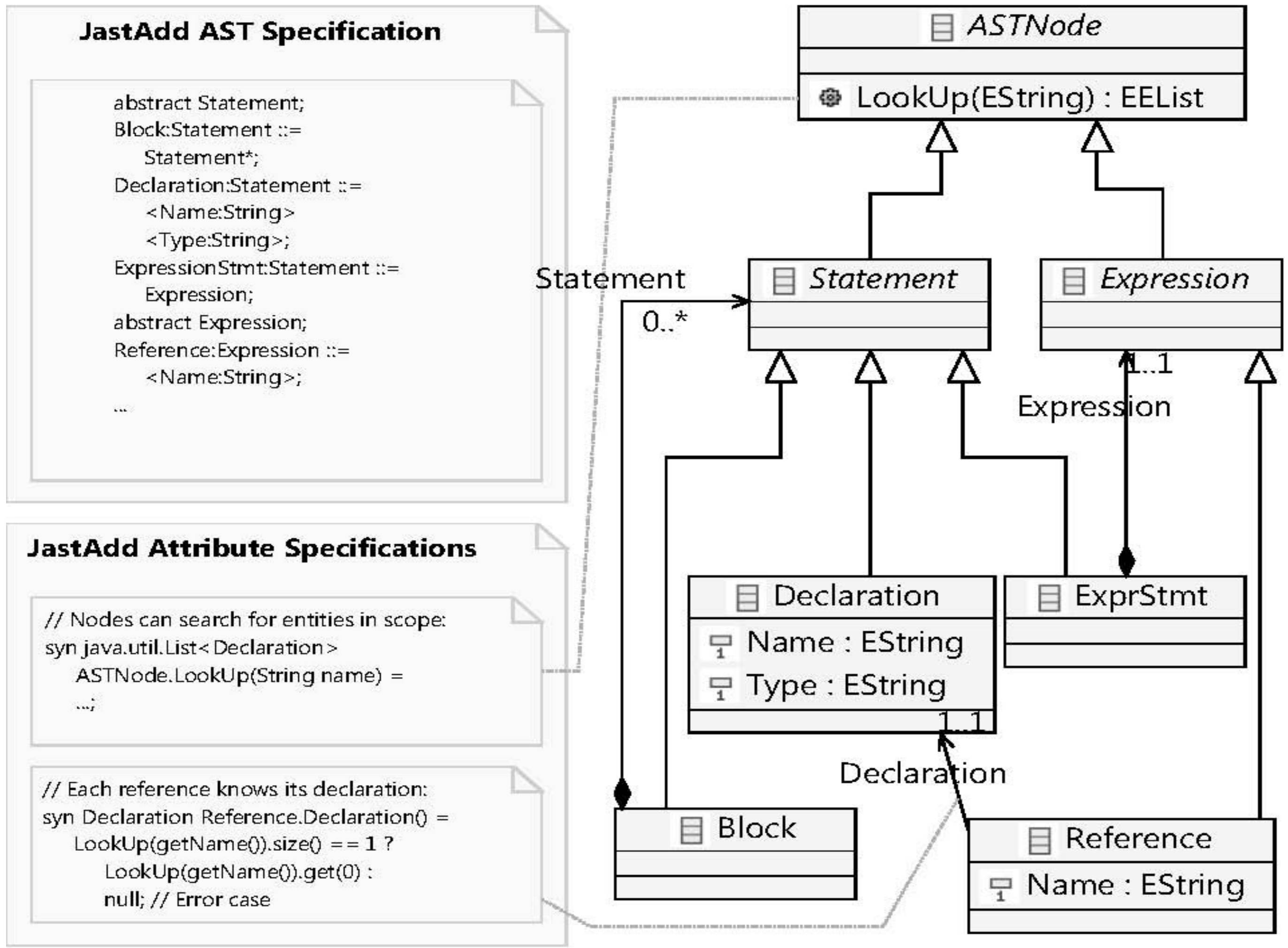


The JastEMF Approach Requires a Ecore-JastAdd Language Mapping (Concept Mapping)

AST node types	EClasses
AST terminal children	EClass non-derived properties
AST non-terminal children	EClass containment references
Synthesized attributes	EClass derived properties
	EClass operations
Inherited attributes	EClass derived properties
	EClass operations
Collection attributes	EClass properties (cardinality > 1)
	EClass non-containment ref. (cardinality > 1)
Reference attributes	EClass non-containment references
Woven methods (Intertype declarations)	EClass operations

The JastEMF Approach Requires a Ecore-JastAdd Language Mapping (Concept Mapping)

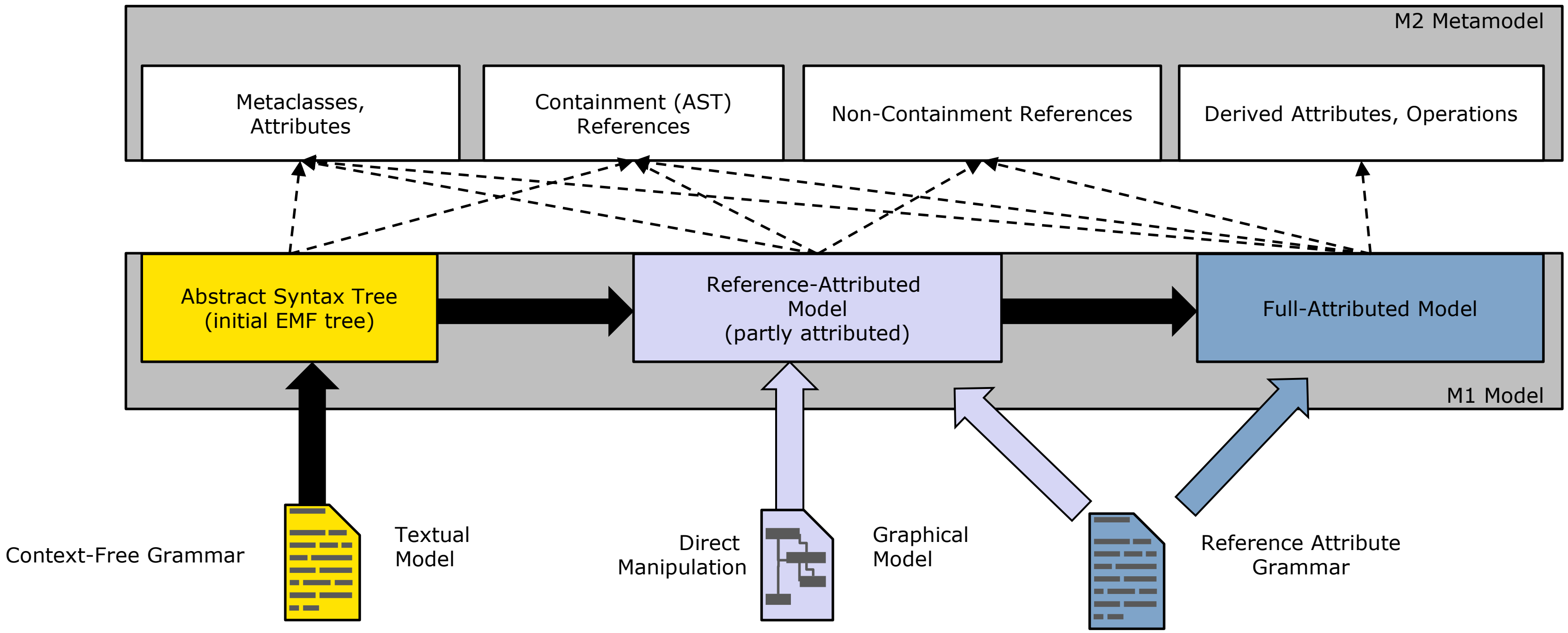
- JastAdd can read EMF files via RelAST importer https://git-st.inf.tu-dresden.de/jastadd/ecore2relast



Stepwise Attribution of Ecore Link-Trees

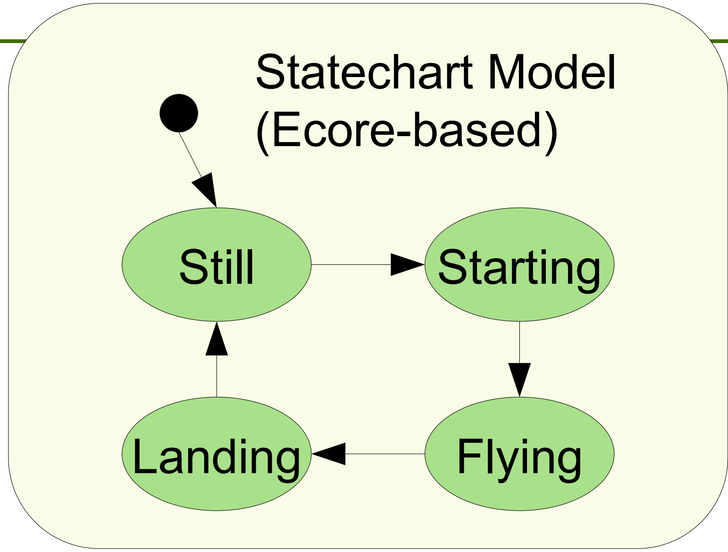
Semantic evaluation can start from (partly) reference-attributed EMOF models

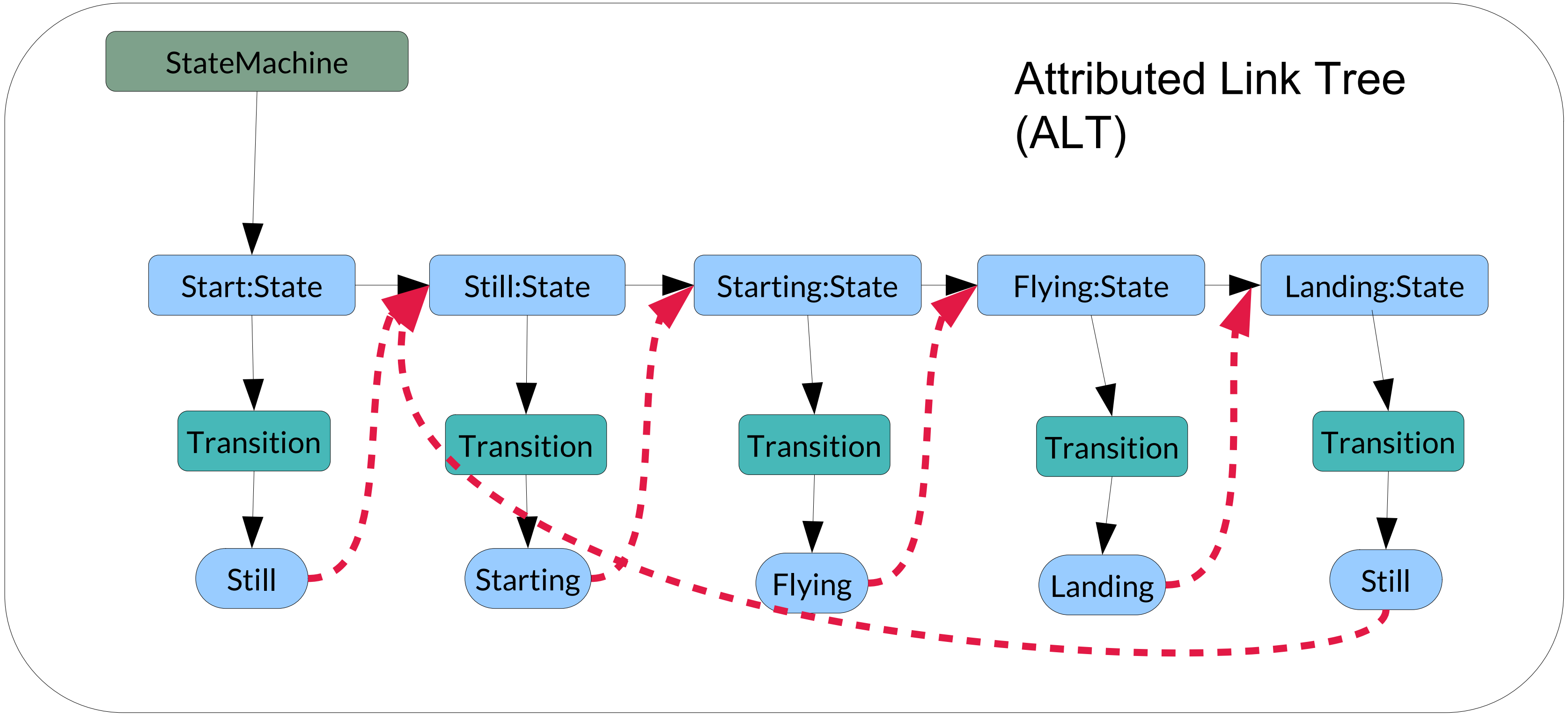
- Non-containment references can have predefined values (e.g. specified by the user in a diagram editor)
- If a value is given: Use it instead of attribute equation



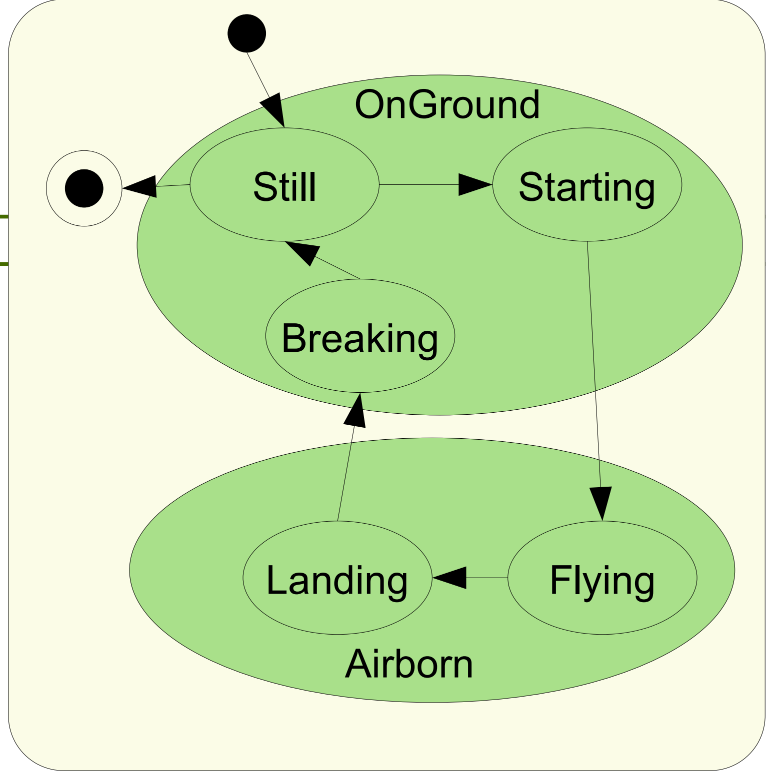
Why Links? (2) Name Analysis in Models

- Models can be represented as link trees
- **Name analysis in models** searches the right definition for a use of a *name* and materializes it as cross-tree link
- This holds for models and programs in *any* language

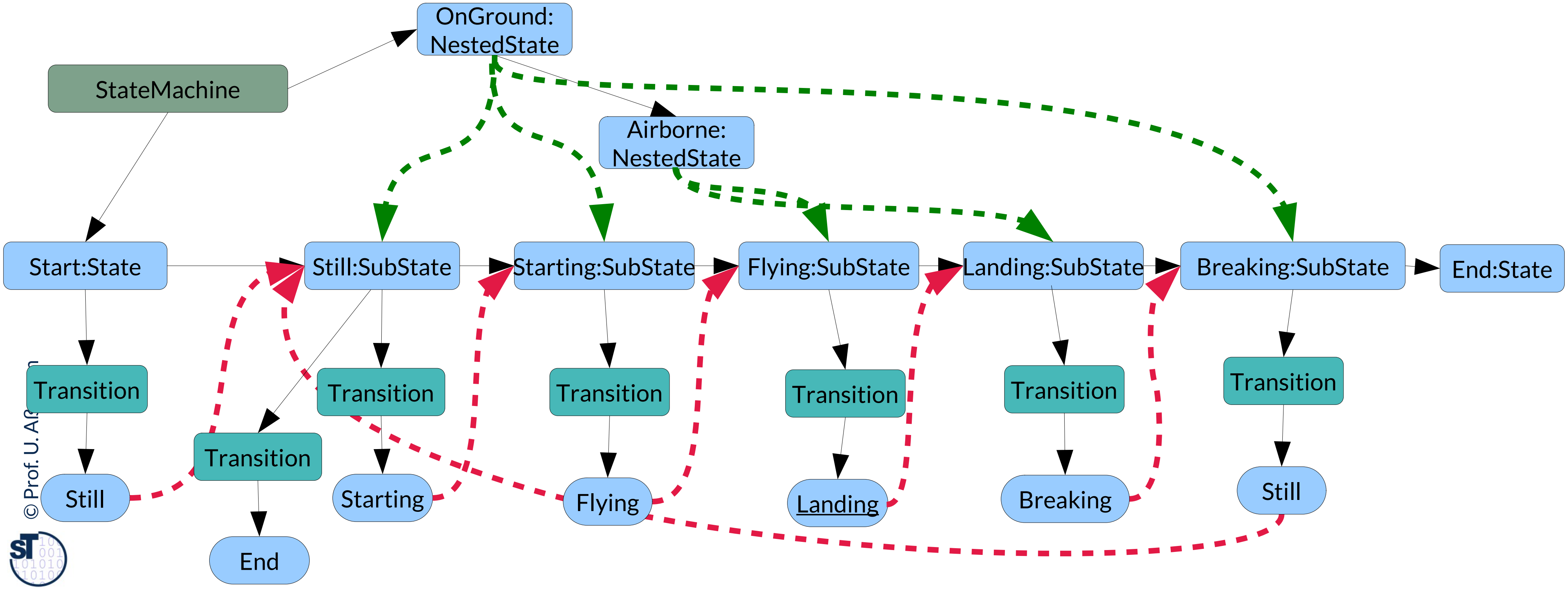




Why Links? (2) Type Analysis in Models

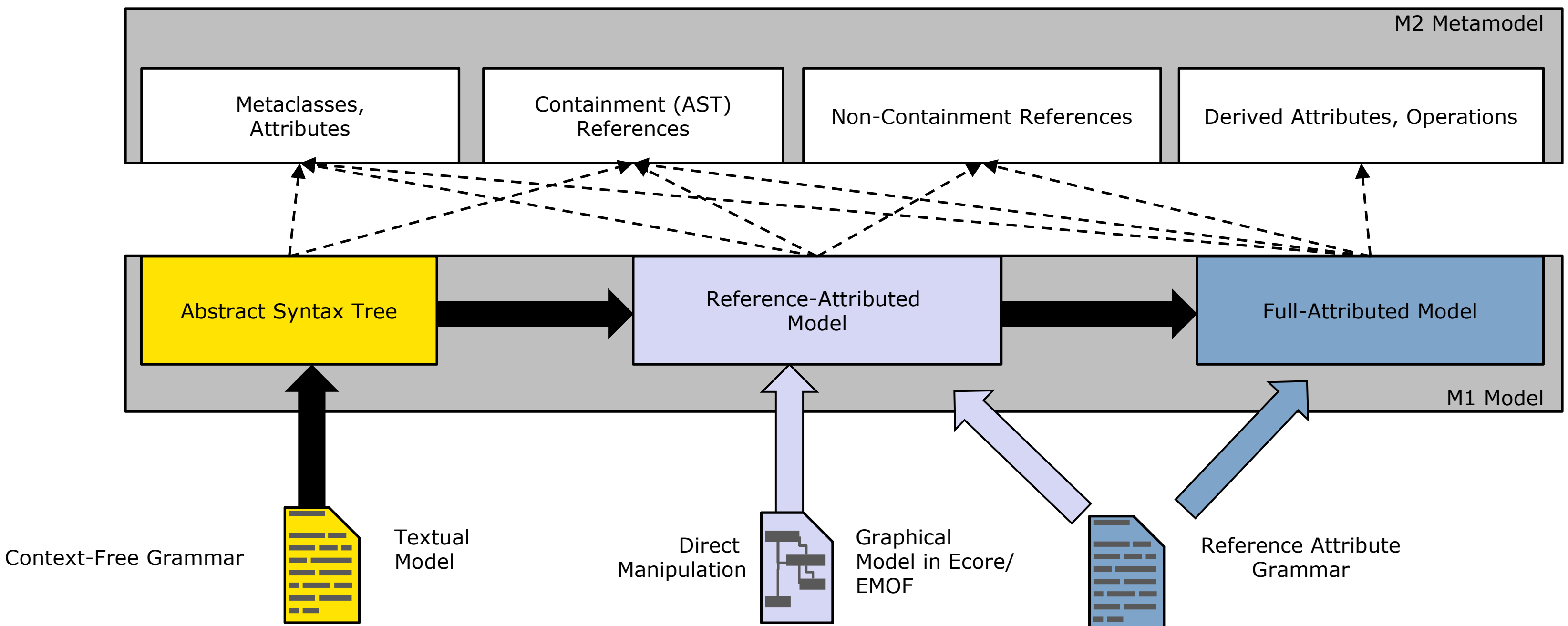


- **Links abbreviate paths to remote nodes**
- Model Element Types can be constrained
- **Type analysis in models** searches the right definition for a use of a *model element type* and materializes it as cross-tree link. Then, wellformedness constraints on the types are checked:
- Ex: forall s: SubState: s has-subtree [0..n] Transition
- forall ns:NestedState: ns has-link-to [1..n] Substate AND ns NOT has-subtree



EMOF as DDL for Reference Attribute Grammars

- Ecore (EMOF) models are ASTs with cross-references and derived information
  - syntactic interface (ASTs); semantic interface (cross-references and derived information)
- Ecore (EMOF) metamodels can be built around a **tree**-based abstract syntax used by
  - Tree iterators, tree editors, transformation tools, interpreters
  - Tools use the tree structure to derive all other information (e.g., resolving cross references, partial interpretation)
  - Graphical editors use the tree structure to manage user created object hierarchies, cross references and values therein and to compute read-only information (e.g., cross references, derived values)



EMOF and Reference Attribute Grammars

- EMOF models are ASTs with cross-references and derived information
- Basically, every form of RAG can be coupled to EMF

AST in Ecore	AST in RAGs	
EClass	AST Node Type	E_{Syn}
EReference[containment]	Nonterminal	
EAttribute[non-derived]	Terminal	

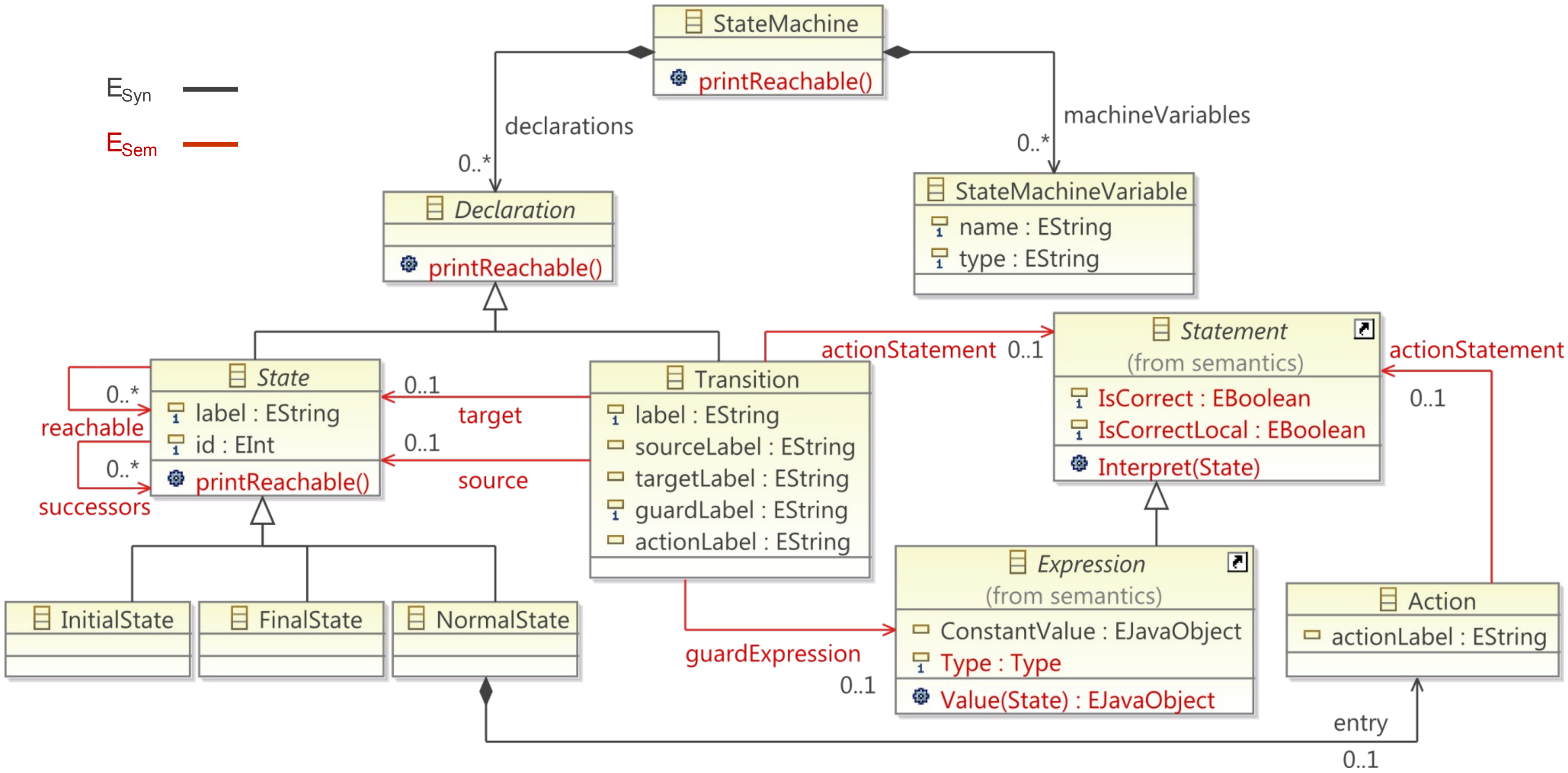
Semantics Interface in Ecore	Semantics in RAGs	
EAttribute[derived]	[synthesized|inherited] attribute	E_{Sem}
EAttribute[derived,multiple]	collection attribute	
EReference[non-containment]	collection attribute, reference attribute	
EOperation[side-effect free]	[synthesized|inherited] attribute	
EReference[containment,derived]	Nonterminal attribute	

24.3 Examples of RAG Applications on EMF

- Writing EMF semantic analyzers with JastEMF
- Writing EMFText Resolvers with JastEMF
- For models and programs

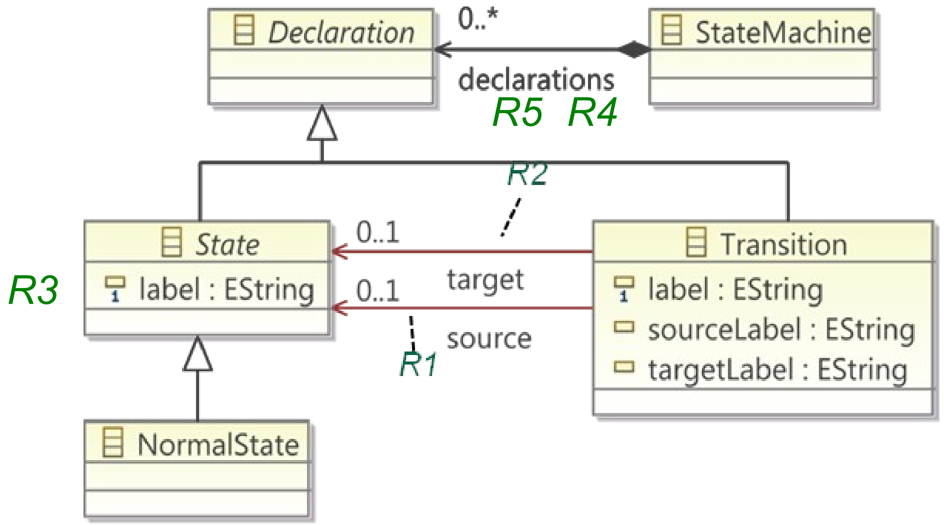
Example 1: Statechart Metamodel in EMOF

Where is the spanning tree?



(Ecore-based, extended version of Statemachine example in Hedin, G.: Generating Language Tools with JastAdd. In: [H09], see also www.jastemf.org)

Example 1: Statechart Metamodel Name Analysis in JastAdd-EMF (JastEMF)



AST specification with RTG (partial):

// Inheritance is ":"

abstract State:Declaration ::= <label:String>;

NormalState:State;

Transition:Declaration ::=<label:String>

<sourceLabel:String><targetLabel:String>;

Attribution example (Specification of abstract interpreter for name analysis):

// synthesized function (bottom-up stencil)

syn lazy State Transition.source() = lookup(getSourceLabel()); // R1

syn lazy State Transition.target() = lookup(getTargetLabel()); // R2

syn **State Declaration.localLookup(String label) = (label==getLabel()) ? this : null;** // R5

// inherited functions (tow-down stencil)

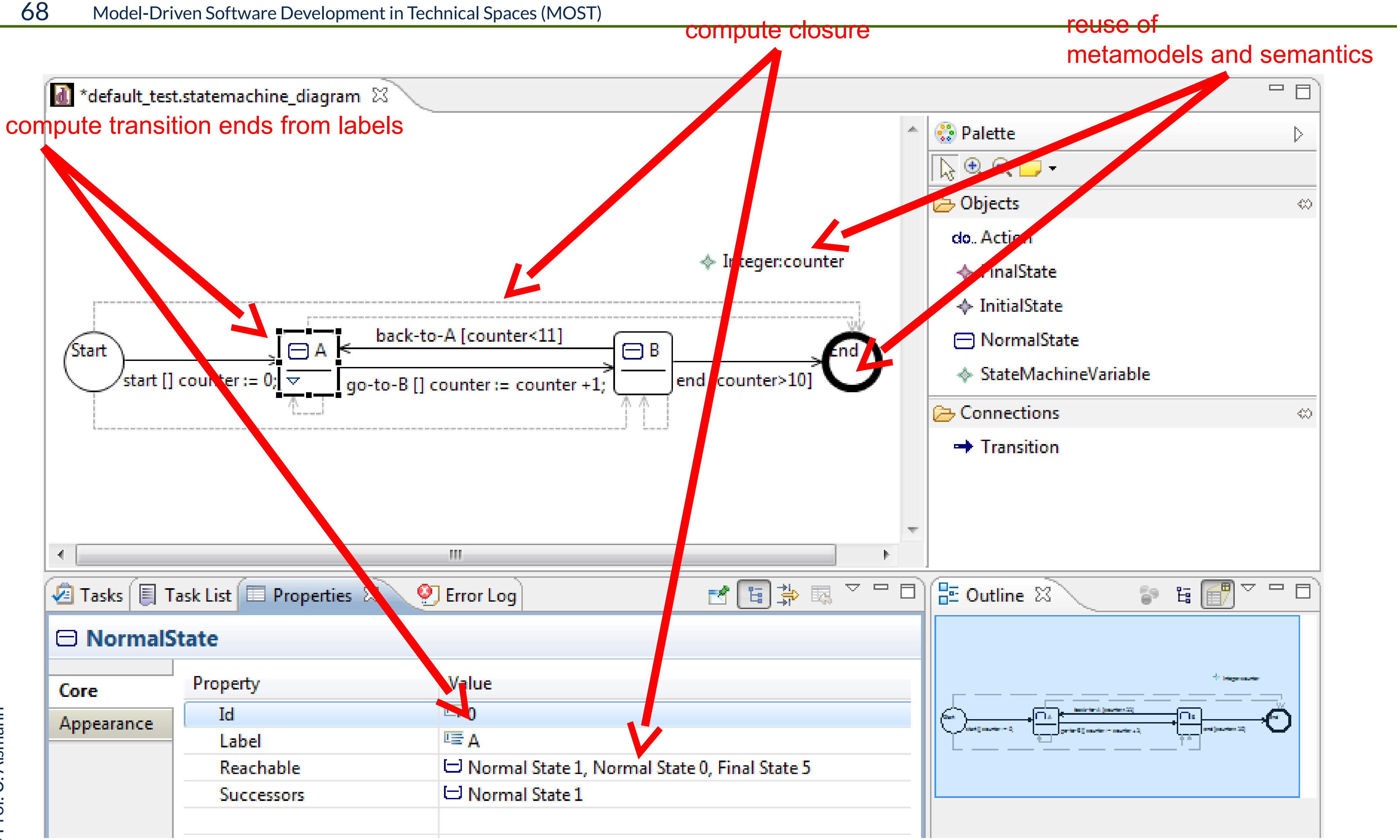
inh State Declaration.lookup(String label); // R3

// Help function

eq StateMachine.getDeclarations(int i).lookup(String label) { … } // R4

(Ecore-based, extended version of Statemachine example in Hedin, G.: Generating Language Tools with JastAdd. In: [H09], see also www.jastemf.org)

Example 1: Generated Statechart Editor with Runtime Semantic Analysis



Restrictions of JastEMF for Model Analysis

RAGs are only well-suited for analysis of models, if the metamodel specifies a wellformed basic tree structure, with overlay links.

The metamodel should not be degenerated which means:

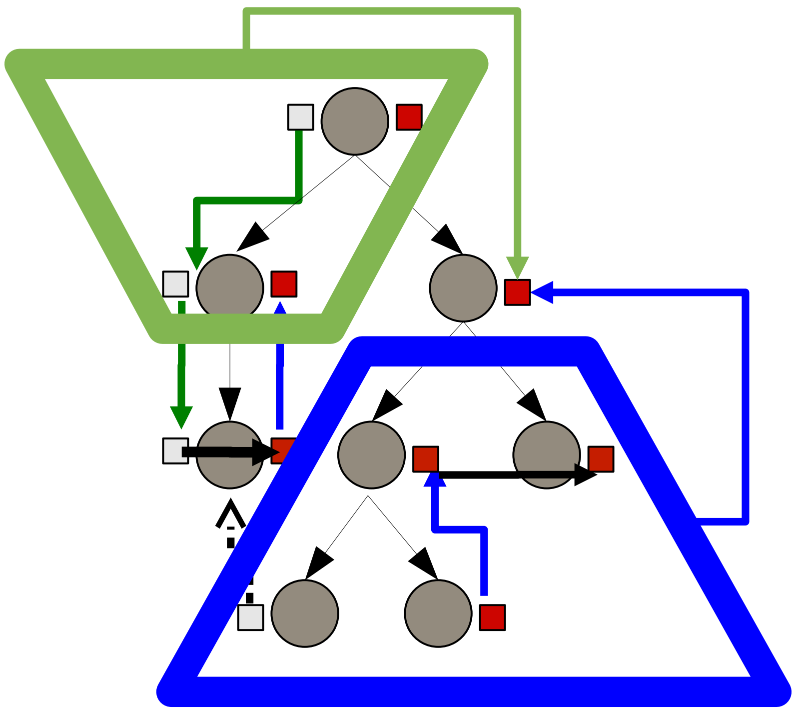
- Nearly no structure modeled at all
- Models have few structural distinguishable entities and/or flat trees
  - Not common in practice (Often a bad modelling indication)
  - Similar to model everything just with collections of collections

24.4 Using Querying as Attributions

- with query stencils

Global Querying with RAGs and Xcerpt

- Usually, attribution functions work locally, Xcerpt queries work globally, searching up and down in the tree
- A ***query attribution function (query stencil)*** queries the tree around the current node
  - `query(): QueryTerm, Tree->Tree`



- **Upward query stencil** (syn, blue): global query downward, result passed **upward**
- **Downward query stencil**: (inh, green): global query upward, result passed **downward**

Ex.: Global Querying with RAGs and Xcerpt

- Query stencils are called with a Query Term from a current node
  - Query stencils do not change the tree
- Suppose a ***query stencil function*** `query(): QueryTerm, Input:Tree->Output:Tree`
  - Input trees are considered as database
  - Output trees can be stored in higher-order (tree) attributes
  - Other output values in normal attributes

```
eq AllConstants.Values() {
   return query(
     “FROM tree {{ Plus(var ConstantValue) }}“,subtree1)
     + query(
     “FROM tree {{ Minus(var ConstantValue) }}“,subtree2)
   ;
}
```

Applications of RAGs for Modeling

- Deep analysis means to link uses and definitions of names, types, packages, classes, methods
- Common metamodels specify **link-tree structures** enriched with semantic interfaces (e.g. EMOF, MOF).
- **RAGs can be used to specify wellformedness (static semantics) for such metamodels**
  - Building up links from pure trees (for name and type analysis)
  - For checking context constraints
  - Completing partially attributed link trees
- JastAdd can be used as RAG tool on Java
- JastEMF (www.jastemf.org): Tool to generate semantic metamodel implementations based on Ecore metamodels and JastAdd Ags.
- Many JastEMF improvements possible
  - Incorporation of incremental AG concepts
  - Better imperative mode (Persistency support for changed attribute values)
  - Incorporation of JastAdd's rewrite capabilities
- Integration of RAG with template processing and global querying posssible

24.5 The Big Picture: The Importance of Link Trees for MDSD Applications

- Link trees, their querying, attribution, and rewriting is very important for an MDSD IDE

Links on the XML Formats of PreeVision

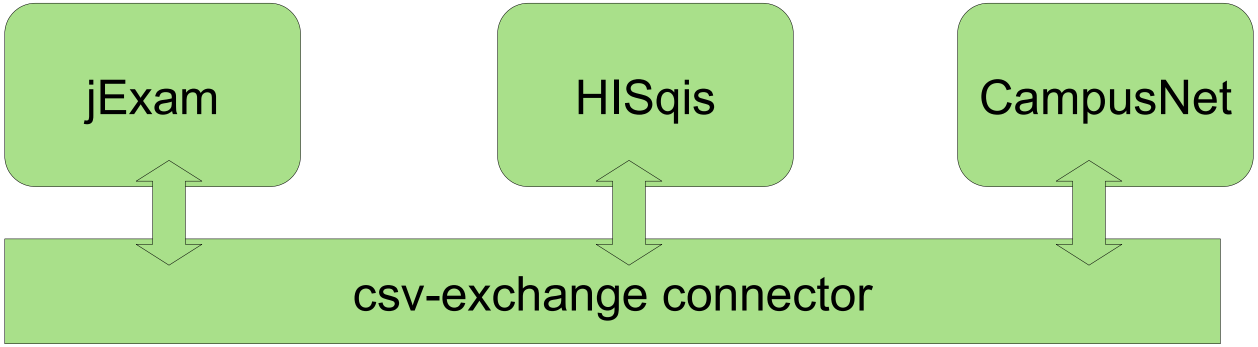
- Excel: https://support.office.com/de-de/article/%C3%9Cberblick-%C3%BCber-XML-in-Excel-f11faa7e-63ae-4166-b3ac-c9e9752a7d80
- RIF: https://en.wikipedia.org/wiki/Requirements_Interchange_Format
- Simulink: http://de.mathworks.com/help/rptgenext/ug/how-to-compare-xml-files-exported-from-simulink-models.html?requestedDomain=www.mathworks.com
- AutoSAR and FIBEX https://vector.com/vi_autosar_de.html
  - https://de.wikipedia.org/wiki/AUTOSAR
  - http://xn--brrkens-b1a.de/publications/pagel_broerkens_ECMDA2006.pdf
  - http://www.elektronikpraxis.vogel.de/embedded-computing/articles/226651/index3.html
  - http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/tools/auxiliary/AUTOSAR_TR_InteroperabilityOfAutosarTools.pdf
  - http://www.sse-tubs.de/publications/Hoe_ASE07.pdf
- LDF http://www.fullconvert.com/XML-to-LDF/

Links on the XML Formats of PreeVision

- ELOG http://www.ecad-if.de/elog.html
- KBL (Kabelbaumliste) http://www.ecad-if.de/kbl.html
- ASCET (von ETAS) http://www.etas.com/data/RealTimes_2010/rt_2010_2_32_de.pdf
  - http://www.etas.com/de/products/ascet_software_products-details.php
  - http://www.file-extensions.org/amd-file-extension-ascet-xml-model-description-file
  - http://www.etas.com/download-center-files/products_ASCET_Software_Products/ETAS_ASCET_6.1_flyer_DE.pdf

Exchange Formats: Use Link-Trees, not CSV!

- csv is notoriously used as exchange format between tools...



- No schema nor metamodel! No name analysis! No format description!
- Use Link-Tree formats, such as XML or Ecore with RAGs that can do the name analysis.
- see part III on Tool Integration.

Comparing Metamodeling and RAGs

Benefits of Metamodelling

Metamodelling is a standardisation process with the following benefits:

- MM 1 Metamodelling Abstraction
- MM 2 Metamodelling Consistency
- MM 3 Metamodel Implementation Generators
- MM 4 Metamodel/Model Compatibility
- MM 5 Tooling Compatibility

However, metamodelling leaks convenient mechanisms for semantics specification.

Benefits of Reference Attribute Grammars (RAGs)

RAGs are very convenient to specify static semantics for <u>tree structure</u> with the following benefits:

AG 1: Declarative Semantics Abstraction

AG 2: Semantics Consistency

AG 3: Semantics Generators

AG 4: Semantics Modularity (Extensibility)

<u>Observation</u>: A combination of MM and RAGs enables *semantics integrated metamodelling* and leads to more successful and reliable tool implementations.

How To Develop an MDSD Application with Link Trees

- Read in XML with XML parser
- Query XML link trees with languages like Xcerpt
- Semantic analysis of the trees with RAG, with languages like JastAdd
- Transform with languages like
    - Xcerpt
    - Stratego (rewriting)
    - RAG tree generation and template expansion

- Problematic: Tool maturity

The End

- Why are XML documents link trees? Is such a document a link term or link tree?
- How does Xcerpt do deep match?
- Explain how Xcerpt transformation expressions filter an input stream and produce an output stream
- Why can RAG work on link trees?
- How to do deep analysis with RAGs?
- How would you analyse the link structure of an XML document?
- How do references in a link tree abbreviate the way from uses to definitions of variables?
- What does name analysis do with regard to the links of a link tree?
- What does type analysis do with regard to the links of a link tree?
- Does a downward query disturb the rest of the attribution in the subtree? (hint: it depends...)

- Many slides are courtesy to Sven Karol and Christoff Bürger. Thanks.

Attribute Grammars

- AG and RAG are a special form of functional programming on trees and link-trees (data-driven programming)
- **Formalism to compute static semantics over (reference-based) syntax trees** [Knuth68]
  - Basis: context-free grammars + attributes + semantic functions
- Evaluation by tree visitors with different visiting strategies
  - Static dependencies: ordered attribute grammars (OAGs)
  - Dynamic dependencies: demand-driven evaluation
- AGs are modular and extensible

- **Improvements**
  - Higher order attribute grammars (HOAGs) [Vogt+89] computing trees, code and models
  - Reference attributed grammars (RAGs) [Hedin00,Boyland05] on link trees
  - Remote-attribute Controlled Rewriting (RACR) [Bürger15] more rewriting

Formal Definition of AG

(Short) Definition (attribute grammar): An attribute grammar (AG) is an 8-tuple $G=(\mathbf{G_0},\mathbf{Syn},\mathbf{Inh},\mathbf{Syn_x},\mathbf{Inh_x},\mathbf{K},\mathbf{\Omega},\mathbf{\Phi})$ with the following components

- $\mathbf{G_0} = (N,\Sigma,P,S)$ a CFG,
- **Syn** and **Inh** the finite, disjoint sets of synthesized and inherited attributes,
- $\mathbf{Syn_x} : N \to P(Syn)$ a function that assigns a set of synthesized attributes to each nonterminal in G_0,
- $\mathbf{Inh_x} : N \to P(Inh)$ a function that assigns a set of inherited attributes to each nonterminal in G_0,
- **K** a set of attribute types/sorts,
- $\mathbf{\Omega} : Inh + Syn \to K$ a function assigning each attribute a $\kappa \in K$,
- $\mathbf{\Phi}$ a set of semantic functions $\varphi_{(p,i,a)}$ with $p \in P$, $i \in \{0,...,n_p\}$, $a \in Syn_x(p_i) \cup Inh_x(p_i)$.