

23. Link-TreeWare: Languages for Link-Tree Querying, Transformations and Rewriting

Prof. Dr. U. Aßmann

Technische Universität Dresden

Institut für Software- und
Multimediatechnik

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

Version 15-1.3, 12.12.15

- 1) DDL for Link Trees
- 2) Analysis with query languages
- 3) Transformation languages for Link Trees
 - 1) Xcerpt
 - 2) Modular Xcerpt
 - 3) Context-sensitive Transformations
- 4) Deep analysis with RAG of textual languages
- 5) Deep analysis of models
- 6) Consequences for MDSD applications



DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

Obligatory Literature

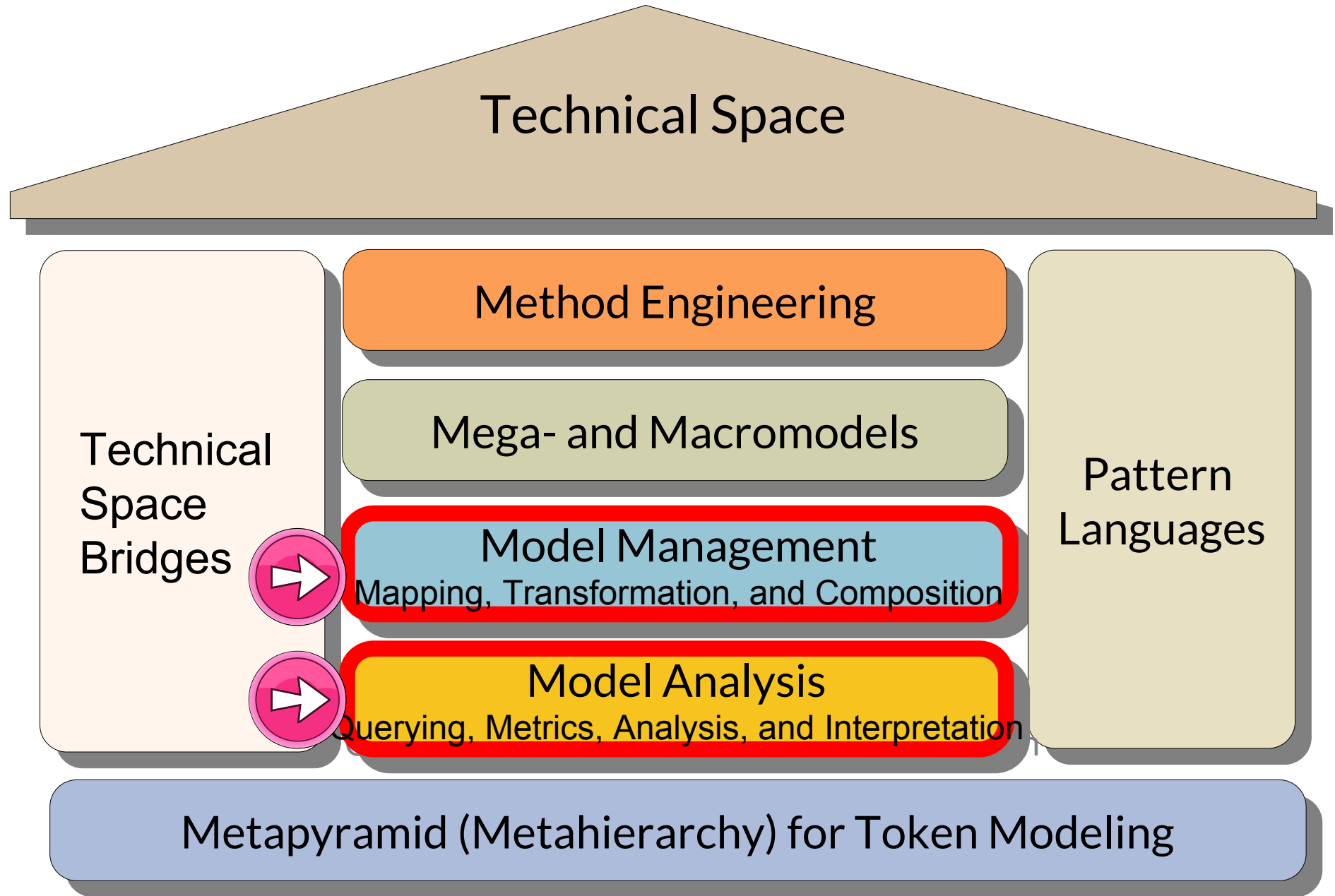
- ▶ Sebastian Schaffert, François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt (2004). In Proc. Extreme Markup Languages.
 - <http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2004-7.pdf>
 - <http://www.rewerse.net/publications/download/REWERSE-RP-2006-069.pdf>

Literature

3 Model-Driven Software Development in Technical Spaces (MOST)

- [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 (July): 627–687.
- [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.
- [Knuth68] Knuth, D. E. 1968. 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.
- [Ekman06] Ekman, Torbjörn. 2006. Extensible Compiler Construction. University of Lund.
- [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.
- [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.
- [Kühnemann+97] Kühnemann, Armin, und Heiko Vogler. 1997. *Attributgrammatiken -- Eine grundlegende Einführung*. Braunschweig/Wiesbaden: Vieweg.
- [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.

Q10: The House of a Technical Space



23.1 Data Definition Languages (DDL) for Link Trees

The basic layer of LinkTreeWare M2



- ▶ A **link tree** is a tree with secondary cross-links. We call the primary tree the **skeleton tree**, and the secondary links form the **overlay graph**.
- ▶ A **map tree (associative tree)** is a tree of maps, i.e., a tree of associative arrays.
- ▶ A **map link tree** is a map tree with (secondary) cross-links.
- ▶ Examples of map link trees:
 - A JSON document
 - An XML document
 - An EMF tree
- ▶ Specification languages (DDL) for link trees *extend RTG* with names (anchors) and links to these names

23.1.1 JSON Associative Map Trees

- ▶ <http://json-schema.org/examples.html>
- ▶ Michael Droettboom, et al. Understanding JSON Schema, Release 1.0. Space Telescope Science Institute, October 12, 2015
 - <http://spacetelescope.github.io/understanding-json-schema/UnderstandingJSONSchema.pdf>



JSON

- ▶ JSON is a family of link map tree languages, mainly for syntax trees, NOT for markup
- ▶ JSON is block-structured with angle brackets; list and set constructors
- ▶ JSON often used as exchange format
- ▶ Metalanguage JSON Schema is a lifted DDL and similar to a RTG with maps
- ▶ Example:

```
// JSON Schema of objects with names and ages
{
  "title": "Example Schema",
  "type": "object",
  "properties": {
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "age": {
      "description": "Age in years",
      "type": "integer",
      "minimum": 0
    }
  },
  "required": ["firstName", "lastName"]
}
```

```
// JSON instance of objects with names and ages
{
  "object": { "firstName": „Uwe“,
              "lastName": "Aßmann",
              "age": { 27 }
            }
  "object": { "firstName": „John“,
              "lastName": "Smith"
            }
}
```

[IETF-JSON] Copyright (c) IETF Trust and the persons identified as authors of the code. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, is permitted pursuant to, and subject to the license terms contained in, the Simplified BSD License set forth in Section 4.c of the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>)


```

{ "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product set",
  "type": "array",
  "items": {
    "title": "Product",
    "type": "object",
    "properties": {
      "id": { "description": "The unique identifier for
        a product",
        "type": "number"
      },
      "name": { "type": "string" },
      "price": { "type": "number",
        "minimum": 0,
        "exclusiveMinimum": true
      },
      "tags": { "type": "array",
        "items": { "type": "string" },
        "minItems": 1,
        "uniqueItems": true
      },
      "dimensions": { "type": "object",
        "properties": { "length": { "type": "number" },
          "width": { "type": "number" },
          "height": { "type": "number" }
        },
        "required": ["length", "width", "height"]
      },
      "warehouseLocation": {
        "description": "Coordinates of the warehouse
          with the product",
        "$ref": "http://json-schema.org/geo"
      }
    },
    "required": ["id", "name", "price"]
  }
}

```

```

// instance
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
      "length": 7.0,
      "width": 12.0,
      "height": 9.5
    },
    "warehouseLocation": {
      "latitude": -78.75,
      "longitude": 20.4
    }
  },
  {
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
      "length": 3.1,
      "width": 1.0,
      "height": 1.0
    },
    "warehouseLocation": {
      "latitude": 54.4,
      "longitude": -32.7
    }
  }
]

```

EMF as Link Tree DDL

- ▶ EMF is link-tree structured because it has a primary aggregation hierarchy, but also allows for general links
- ▶ A skeleton tree can be identified in all EMF link trees

23.1.2 Technical Space Link-Treeware with Metalanguages for XML

Data in tree format with overlaying links (references)

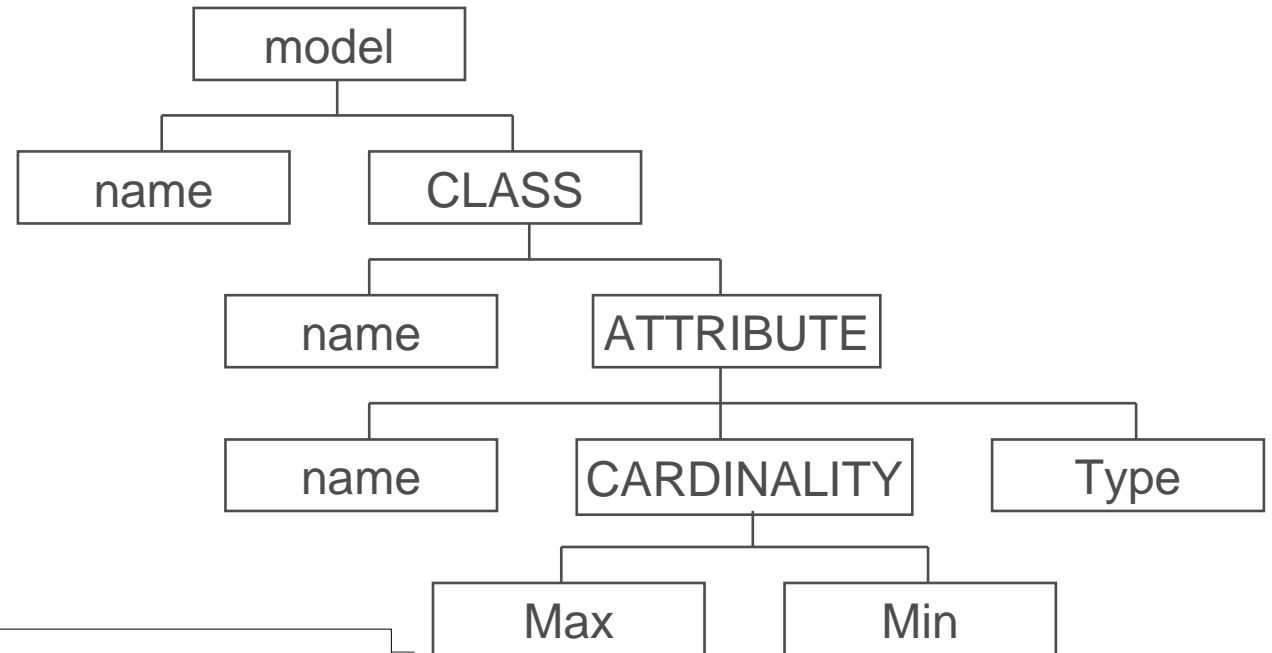


DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

- ▶ XML is a family of link-tree languages, mainly for
 - Markup of data
 - Syntax trees
- ▶ <http://www.w3.org/XML>
- ▶ XML is block-structured with “tag” brackets; only list, no set constructor
- ▶ XML often used as exchange format
- ▶ XML metamodels on M2 can be defined with several metalanguages on M3
 - Document Type Definitions (DTD): special DDL
 - XML Schema Definition (XSD) is a lifted DDL and similar to a RTG with maps
 - RelaxNG: special DDL

Document Type Definition (DTD) for XML

- ▶ A **DTD** is a simple metalanguage for XML
- ▶ Based on regular tree grammars (RTG)



```
<! ELEMENT model (name, CLASS)>
<! ELEMENT CLASS (name, ATTRIBUTE*)>
<! ELEMENT name #PCDATA REQUIRED>
<! ELEMENT ATTRIBUTE (name, CARDINALITY?, Type?)>
<! ELEMENT CARDINALITY (Min, Max)>
<! ELEMENT Min (#PCDATA) REQUIRED>
<! ELEMENT Max (#PCDATA)>
<! ELEMENT Type (#PCDATA)>
```

Example for an XML Document Instance

- ▶ Uses all ELEMENTs of a XSD or DTD as “tags” in an XML model

```
<? xml version = „1.0“?>
<! DOCTYPE oomodel SYSTEM „oom.dtd“>
<model>
  <name> „Model 1“ </name>
  <CLASS>
    <name> „Class 1“ </name>
    <ATTRIBUTE>
      <name> „attribute 1“ </name>
      <CARDINALITY>
        <Min> „1“ </Min>
        <Max> „1“ </Max>
      </CARDINALITY>
      <Type> „Class 1“ </Type>
    </ATTRIBUTE>
  </CLASS>
</model>
```

Ex.: DTD for Property Lists in XML

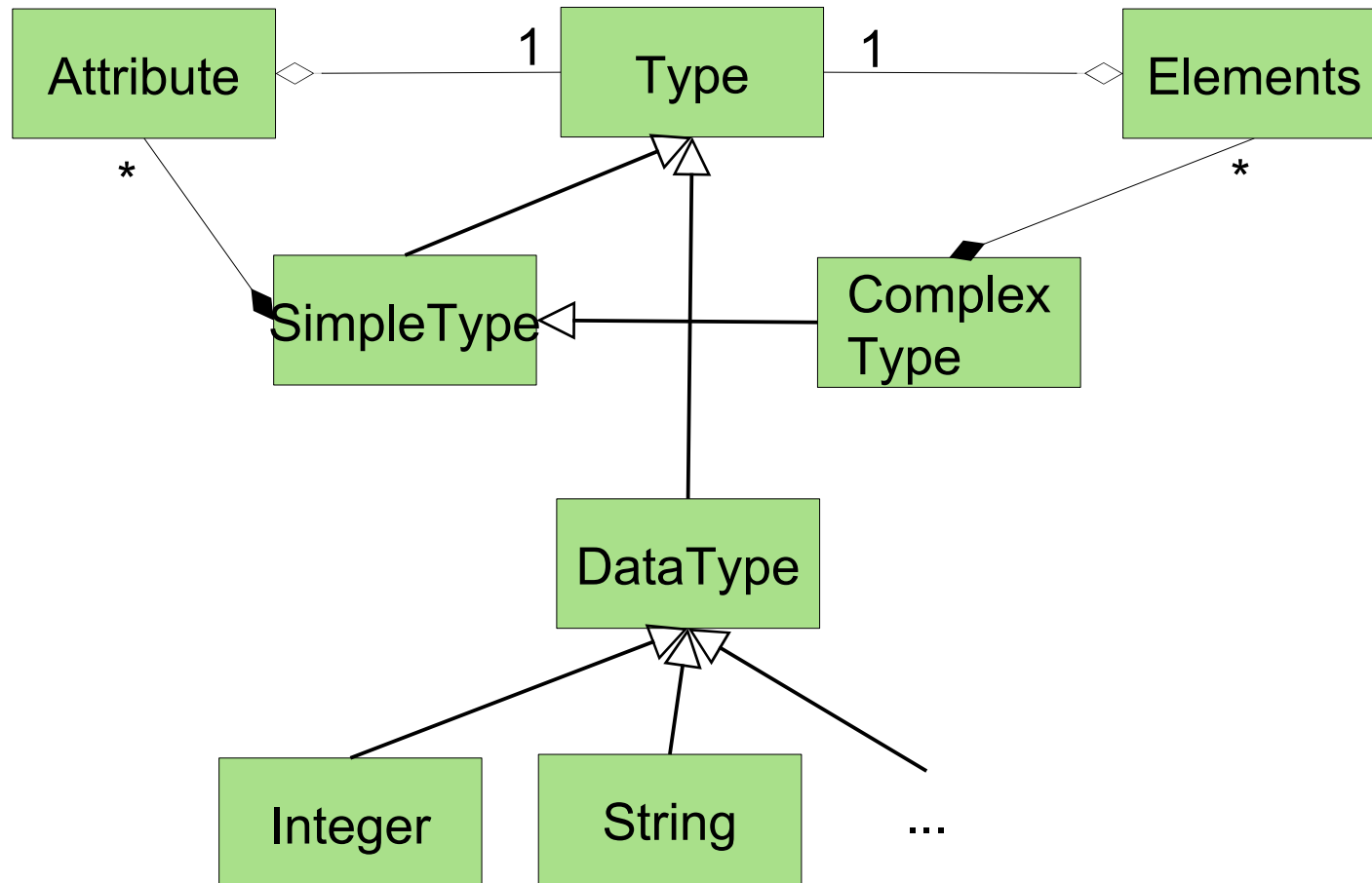
```
<!ENTITY % plistObject "(array | data | date | dict | real | integer | string |
    true | false )" >
<!ELEMENT plist %plistObject;>
<!ATTLIST plist version CDATA "1.0" >
<!-- Collections -->
<!ELEMENT array (%plistObject;)*>
<!ELEMENT dict (key, %plistObject;)*>
<!ELEMENT key (#PCDATA)>
<!-- Primitive types -->
<!ELEMENT string (#PCDATA)>
<!ELEMENT data (#PCDATA)> <!-- Contents interpreted as Base-64 encoded -->
<!ELEMENT date (#PCDATA)> <!-- Contents should conform to a subset of ISO 8601
    (in particular, YYYY '-' MM '-' DD 'T' HH ':' MM ':' SS 'Z'.  Smaller units
    may be omitted with a loss of precision) -->
<!-- Numerical primitives -->
<!ELEMENT true EMPTY> <!-- Boolean constant true -->
<!ELEMENT false EMPTY> <!-- Boolean constant false -->
<!ELEMENT real (#PCDATA)> <!-- Contents should represent a floating point
    number matching ("+" | "-")? d+ (".d*")? ("E" ("+" | "-") d+)?
    where d is a digit 0-9. -->
<!ELEMENT integer (#PCDATA)> <!-- Contents should represent a (possibly
    signed) integer number in base 10 -->
```

XML Schema Definition (XSD)

16

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ XSD is a metalanguage (lifted metamodel) for the definition of XML-dialects
- ▶ For definition of the valid “tags” and their structure
 - XML-Syntax
- ▶ MOF-metamodel von XSD:



XML Example

17

Model-Driven Software Development in Technical Spaces (MOST)

```
<treatment>
```

```
  <patient insurer="1577500"nr='0503760072' />  
  <doctor  city    ="HD"          nr='4321' />
```

simple

```
  <service>  
    <mkey>1234-A</mkey>  
    <date>2001-01-30</date>  
    <diagnosis>No complications.  
  </diagnosis>  
</service>
```

complex

```
</treatment>
```

[W. Löwe, Linnaeus Universitet Växjö]

Example: Definition of Simple and Complex Tag Types with XML Schema (XSD)

```
<simpletype name='mkey' base='string'>
  <pattern value='[0-9]+(-[A-Z]+)?' />
</simpletype>

<simpletype name='insurer' base='integer'>
  <precision value='7' />
</simpletype>

<simpletype name='myDate' base='date'>
  <minInclusive value='2001-01-01' />
  <maxExclusive value='2001-04-01' />
</simpletype>

<complextype name='treatment'>
  <element name='patient' type='patient' />
  <choice>
    <element ref='doctor' />
    <element ref='hospital' />
  </choice>
  <element ref='service' maxOccurs='unbounded' />
</complextype>
```

Example:

XML Schema Attributes

```
<complexType name='patient' content='empty'>
  <attribute ref='insurer' use='required' />

  <attribute name='nr' use='required'>
    <simpletype base='integer'>
      <precision value='10' />
    </simpletype>
  </attribute>

  <attribute name='since' type='myDate' />
</complexType>
```

23.2. Analysis on Link Trees



Analysis with Query Languages (Anfragesprachen)

- ▶ Query languages (QL) are used for analysis:
 - Match patterns in models or entire repositories
 - Query reachability of model elements
 - Count elements (metrics)
 - Filter streams
- ▶ Applications:
 - QL with transformations are useful for writing Filters
 - Consistency Checking in Trees
 - Slicing of trees (view construction)

23.2.1 XQuery

The standard from W3C



DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

Xquery

- ▶ <http://www.w3.org/XML/Query/>
- ▶ Standard of W3C for XML queries
- ▶ Patterns and query expressions are embedded in loops over input streams
- ▶ Output can be embedded into XML models or HTML pages [\[http://www.w3.org/TR/xquery/\]](http://www.w3.org/TR/xquery/)

Input stream

```
for $e in $employees
order by $e/salary
descending
return $e/name
```

Process

Output stream

```
for $b in $books/book
stable order by $b/title
collation
"http://www.example.org/collations/fr-ca",
$b/price descending empty least
return $b
```

Hamlet (Shakespeare) in XML Markup

```
<?xml version="1.0"?>
<!DOCTYPE PLAY SYSTEM "play.dtd">
<PLAY> <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE> <FM>
  <P>Text placed in the public domain by Moby Lexical Tools, 1992.</P>
  <P>SGML markup by Jon Bosak, 1992-1994.</P> <P>XML version by Jon Bosak, 1996-1998.</P>
  <P>This work may be freely copied and distributed worldwide.</P>
</FM>
<PERSONAE>
<TITLE>Dramatis Personae</TITLE>
<PERSONA>CLAUDIUS, king of Denmark. </PERSONA>
<PERSONA>HAMLET, son to the late, and nephew to the present king.</PERSONA>
<PERSONA>POLONIUS, lord chamberlain. </PERSONA>
<PERSONA>HORATIO, friend to Hamlet.</PERSONA>
</PERSONAE>
<ACT><TITLE>ACT I</TITLE>
<SCENE><TITLE>SCENE I. Elsinore. A platform before the castle.</TITLE>
<STAGEDIR>FRANCISCO at his post. Enter to him BERNARDO</STAGEDIR>
<SPEECH> <SPEAKER>BERNARDO</SPEAKER> <LINE>Who's there?</LINE> </SPEECH>
<SPEECH> <SPEAKER>FRANCISCO</SPEAKER> <LINE>Nay, answer me: stand, and unfold yourself.</LINE> </SPEECH>
<STAGEDIR>Exeunt</STAGEDIR>
</SCENE>
</ACT>
<ACT><TITLE>ACT II</TITLE>
...
</ACT>
</PLAY>
```


Xquery is a Mixed Language

The following script produces a list of speakers of the hamlet plot

```
<html><head/><body>
```

```
{
```

```
for $act in doc("hamlet.xml")//ACT
let $speakers := distinct-values($act//SPEAKER)
return
```

```
<div>
```

```
<h1>{ string($act/TITLE) }</h1>
```

```
<ul>
```

```
{
```

```
for $speaker in $speakers
return <li>{ $speaker }</li>
```

```
}
```

```
</ul>
```

```
</div>
```

```
}
```

```
</body></html>
```

```
<?xml version="1.0"?>
```

23.2.2 The Query and Term Transformation Language Xcerpt

A modern, declarative query and transformation
language for link trees in the XML technical space

Xcerpt combines a DQL and a DTL



Literature - Modular Xcerpt

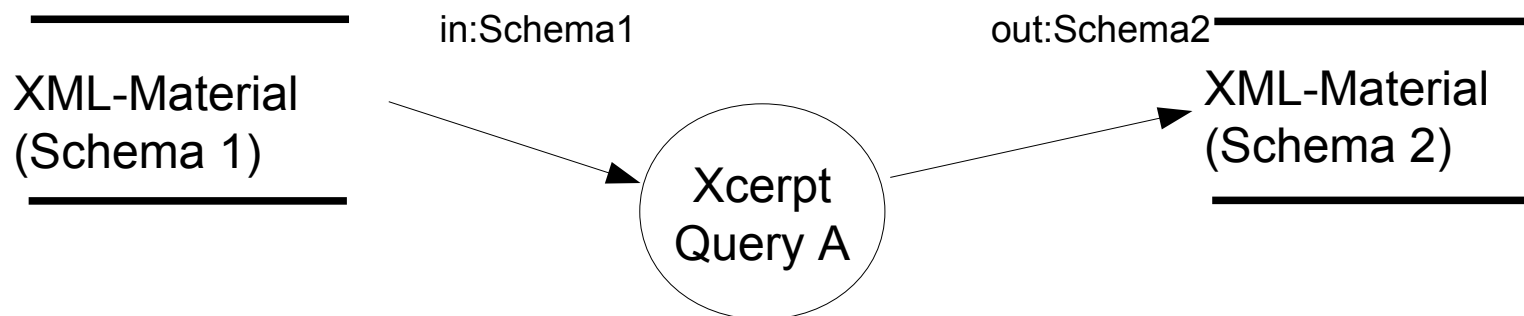
27

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Xcerpt prototype compiler: <http://sourceforge.net/projects/xcerpt>
- ▶ Sebastian Schaffert. Xcerpt: A Rule-Based Query and Transformation Language for the Web. PhD Thesis, Institute for Informatics, University of Munich, 2004.
- ▶ Sebastian Schaffert, François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt (2004) In Proc. Extreme Markup Languages
 - <http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2004-7.pdf>
- ▶ U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and J. Johannes. Modular web queries from rules to stores. In 3rd International Workshop On Scalable Semantic Web Knowledge Base Systems.
- ▶ Uwe Aßmann, Andreas Bartho, Wlodek Drabent, Jakob Henriksson and Artur Wilk. Composition Framework and Typing Technology tutorial In Rewerse I3-d14
 - <http://rewerse.net/deliverables/m48/i3-d14.pdf>
- ▶ Jakob Henriksson and Uwe Aßmann. Component Models for Semantic Web Languages. In Semantic Techniques for the Web. Lecture Notes in Computer Science 5500. Springer Berlin / Heidelberg, ISSN 0302-9743, 2009
 - <http://springerlink.metapress.com/content/x8q1m87165873127/?p=edfdbbaec29743d59da1cd6f1ea50826&pi=4>
- ▶ Artur Wilk. Xcerpt web site with example queries.
 - <http://www.ida.liu.se/~artwi/XcerptT>

Xcerpt: A Modern Web Query Language

- ▶ Xcerpt is a modern, pattern-based query language for XML formatted data
 - Terms, trees, link trees, link terms (JSON and XML terms)
 - Patterns match data w.r.t. document structure
 - Fully declarative, in contrast to Xquery
 - Rule-based, declarative style of Logic Programming (LP)
 - Much more flexible than XPath, which supports only path-based selection
- ▶ Xcerpt is also a transformation language in form of a term rewrite system (Termersetzungssystem):
 - Separate query terms (left-hand side) and construct terms (right-hand side) not like in XQuery)
 - it has “Construct terms” to simplify creation of new documents
- ▶ Xcerpt is stream-based: processes read and write streams
 - Xcerpt can be used as generator and transformer in DFD



Xcerpt Data Terms (XML Terms)

- ▶ Xcerpt data terms represent XML link-terms with nice syntax
- ▶ Basic constructors for data terms:
 - **exact description of a collection of children:**
 - ordered list [...],
 - unordered set {...}
 - **partial description of a collection of children:**
 - ordered partial list [[...]]
 - unordered partial set {{...}}
 - **references/links:**
 - key id@, keyref ^id

```
<book><title>The Last Nizam</title></book>
```

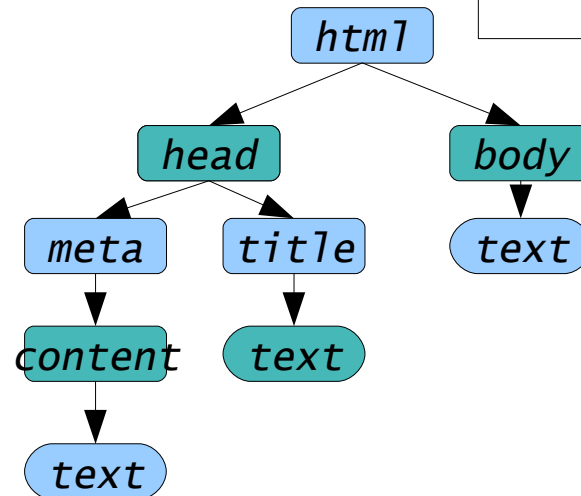
equivalent
to:

```
book [ title [ "The Last Nizam" ] ]
```

Xcerpt Data Terms

```
html [  
  head [  
    meta [  
      content {"text/html"}  
    ]  
    title ["Website"]  
  ],  
  body ["content"]  
]
```

```
<html>  
  <head>  
    <meta content="text/html"/>  
    <title> Website </title>  
  </head>  
  <body>  
    content  
  </body>  
</html>
```



Xcerpt Query Terms are Data Terms with Variables

- ▶ A **query term** is a term pattern *containing variables* (noted in uppercase letters) over XML data, underspecified data terms:

- Ordered matching: `data [term [... X]]`
- No order matching: `data { term { ... X } }`
- Ordered partial matching: `[[X ...]]`
- Unordered partial matching: `{{ ... X }}`
- Queries connect query terms with logical expressions:
`and { ... }, or { ... }`
- Variables can unify to subterms

- ▶ Query terms are data terms with variables prefixed by keyword “var”

```
book [ title [ var X ] ]
```

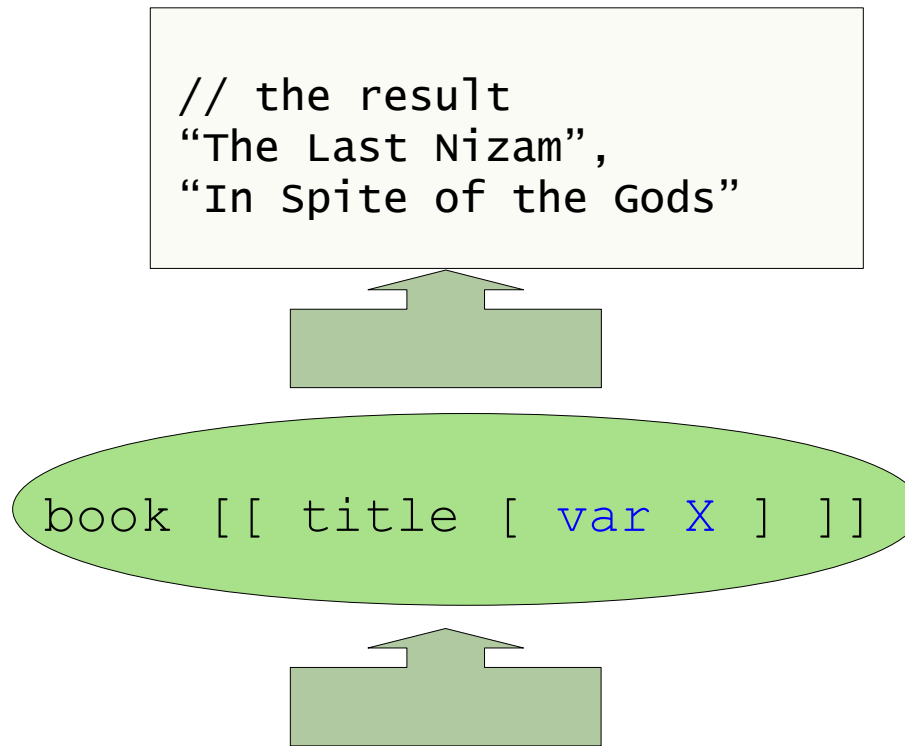
```
// the data base
bib [
  book [ title [ “The Last Nizam” ], author [ “John Zubrzycki” ] ],
  book [ title [ “In spite of the Gods” ], author [ “Edward Luce” ] ]
]
```

Xcerpt Query Terms

32

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ **Query terms** are data terms with variables prefixed by keyword “var”
- ▶ Construct Terms also use variables



```
// the data base
book [ title [ "The Last Nizam" ], author [ "John Zubrzycki" ] ],
book [ title [ "In spite of the Gods" ], author [ "Edward Luce" ] ]
```


Deep Partial Unordered Matching with Xcerpt Query Terms

- ▶ A **query term** (match expression, left-hand side of a rule) is a data term with variables and partial matching

```
library {{  
  book {{  
    // var Author sets a label on a data term  
    var Author -> author {{ surname { "Aßmann" } }},  
    // var Title matches a subterm  
    title [ var Title ]  
  }}  
}}
```

- ▶ Query matches all books with at least one author "Aßmann"
 - assigns the matched authors to variable Author
 - assigns the matched book titles to variable Title
- ▶ Produces a stream of a pair of variables (Author, Title)



23.3 Link-Tree- and XML-based Data Transformation Languages (DTL)

Text, XML, Term, and Graph Rewriting

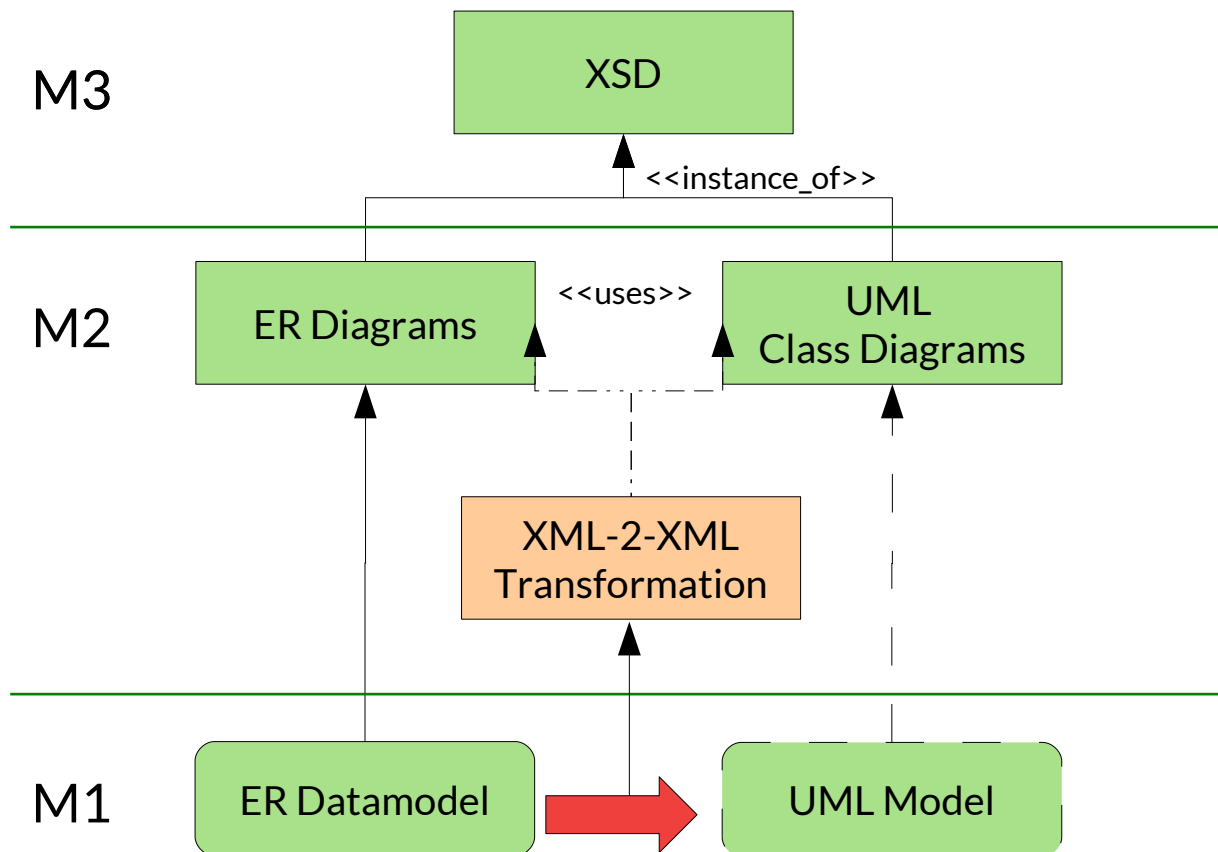


Tree, Term, Link Tree, and XML transformations

35

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Model transformations defined in Layer M_{i+1} specify how to transform models on M_i
 - Source and target metamodel
- ▶ **Benefit:** Transformation can be reused for all models, which are instances of the source meta-model



- ▶ With a data manipulation language (DML, Datenmanipulationssprachen) data is transformed from one form to another
- ▶ **Declarative DML (data transformation languages, Datentransformationssprachen, DTL)** use rules to describe transformations.
 - Specifications consist of rules, grouped in *rule sets* or *rule modules*
 - Term rewrite rules (Termersetzungssregeln) transform trees, terms and dags
 - Graph rewrite rules transform graphs and models
 - In *free rewriting*, the control-flow specification is not necessary
 - In *strategic rewriting*, a strategy (higher-order function) controls the rewriting
 - In *programmed rewriting*, a program or workflow controls rewriting
- ▶ **Imperative DML (allgemeine DML)** know states, side effects, heaps.
- ▶ Examples of declarative DML (DTL):
 - Xquery
 - Xcerpt als Strom-Manipulationssprache
 - XGRS and Fujiaba (on graphs)

Term and Tree Rewrite Systems (Termersetzungssysteme, TRS)

- ▶ **Link-Tree Rewrite Systems (LTRS)** work on trees with links
 - e.g., XML-trees (**XML-Rewrite Systems**)
- ▶ Use:
 - Links are used as abbreviations to remote siblings in the tree
 - Links must be controlled on consistency
- ▶

23.3.1 Transformation with Xcerpt



Xcerpt Construct Terms

- ▶ **Construct Terms** (transformation expressions, right-hand sides of a rule) construct arbitrary structured XML data
 - access data from variables bound by query terms
 - aggregate/re-group data
 - can only have single brackets (no optional content)

- ▶ Constructing one title/author pair in a result tag:

```
result {  
    var Title, var Author  
}
```

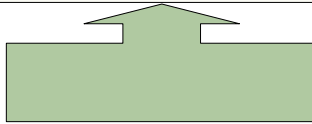
- ▶ Constructing a complete books result list grouped by full author name:

```
booklist {  
    all books {  
        all var Author,  
        var Title  
    }  
}
```

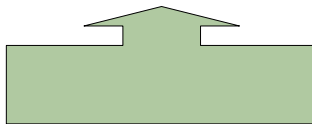
Xcerpt Transformation Rules

- ▶ Combine Query and Construct Terms via common variables

```
// the result  
title [ book [ "The Last Nizam" ] ],  
title [ book [ "In Spite of the Gods" ] ]
```



```
FROM book [[ title [ var X ] ]]  
CONSTRUCT title [ book [ var X ] ]
```



```
// the data base  
book [ title [ "The Last Nizam" ], author [ "John Zubrzycki" ] ],  
book [ title [ "In Spite of the Gods" ], author [ "Edward Luce" ] ]
```


Xcerpt Programs are Rule Sets

- ▶ Xcerpt programs consist of rules and data-terms
 - 1+ goal rules
 - 0+ construct-query rules
 - 0+ data-terms

- ▶ *Construct rules (transformation rules)*: produce intermediate

CONSTRUCT <head> **FROM** <body> **END**

FROM <body> **CONSTRUCT** <head> **END**

Result schema
of a rule

Goal schema

- ▶ *Goal rules*: final output

GOAL <head> **FROM** <body> **END**

- Where <head>: *construct term*; <body>: *query*

```
CONSTRUCT
  construct term
FROM
  query term
END
```

Simple Xcerpt Program

- ▶ Matching query → variable bindings
→ apply bindings to construct term

```
CONSTRUCT
  titles [
    all title [ var Title ]
  ]
FROM
  bib {{
    book {{
      title [ var Title ],
    }} }}
END
```

produce →

```
titles [
  title [
    "The Last Nizam" ],
  title [
    "In Spite of the Gods" ]
]
```

query ↑

```
// the data base
bib [
  book [ title [ "The Last Nizam" ], author [ "John Zubrzycki" ] ],
  book [ title [ "In Spite of the Gods" ], author [ "Edward Luce" ] ]
]
```



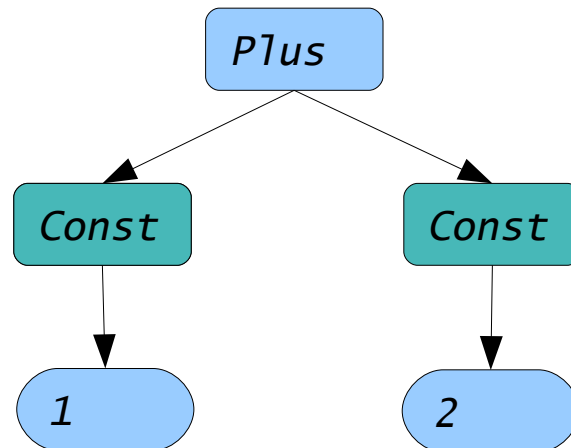
23.3.2 Code Transformations with Xcerpt Term Rewriting



Xcerpt Data Terms for Representing Expression Code

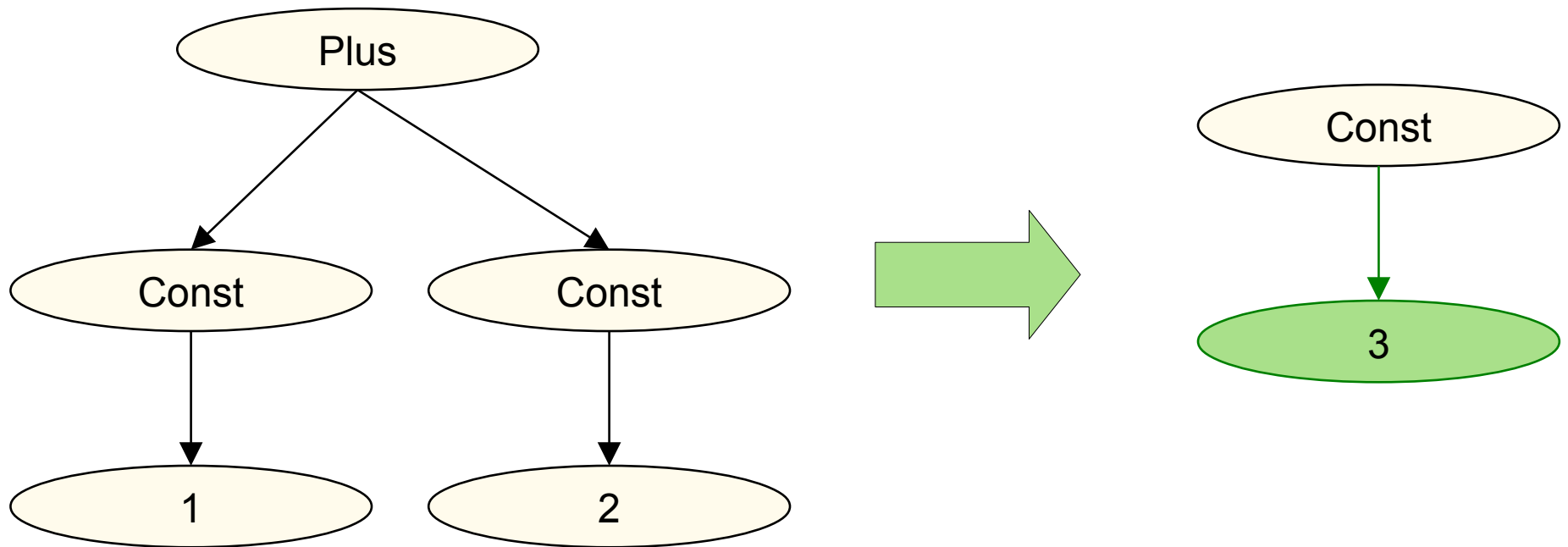
```
Plus [  
  Const [ 1 ],  
  Const [ 2 ]  
]
```

```
<Plus>  
  <Const> 1  
</Const>  
  <Const> 2  
</Const>  
</Plus>
```



Constant Folding (Static Evaluation)

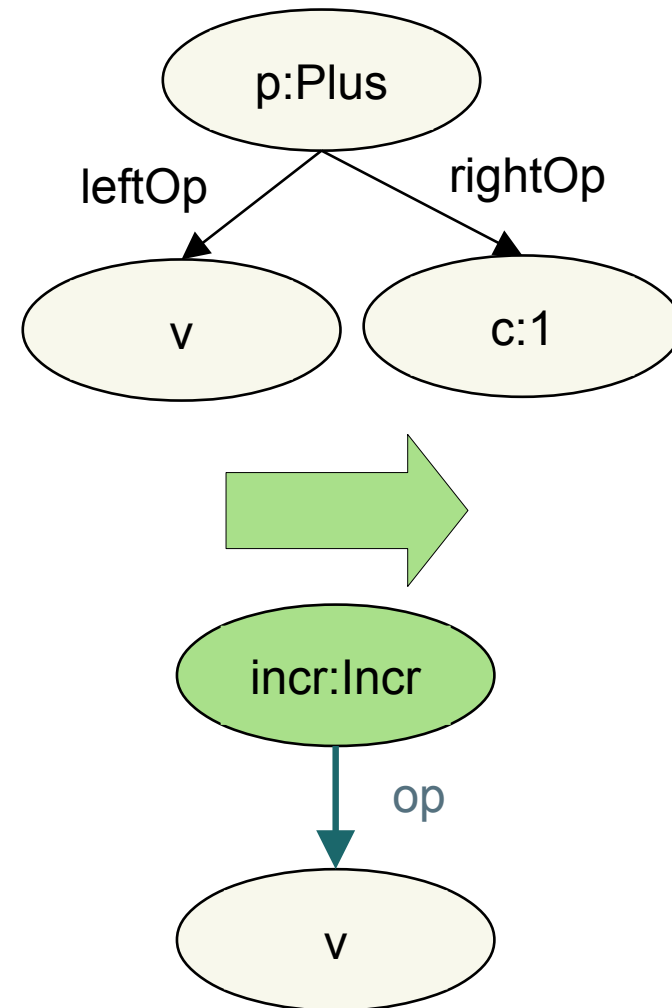
- ▶ A **local rewriting (context-free rewriting)** matches a weakly connected left-hand side graph with a redex, independently of a context.
 - Matching of one redex can be done in constant time
- ▶ Subtractive because redexes are destroyed



FROM Plus [Const [1], Const [2]] **CONSTRUCT** Const [3]

Context-Free Local Rewritings: Operator Strength Reduction

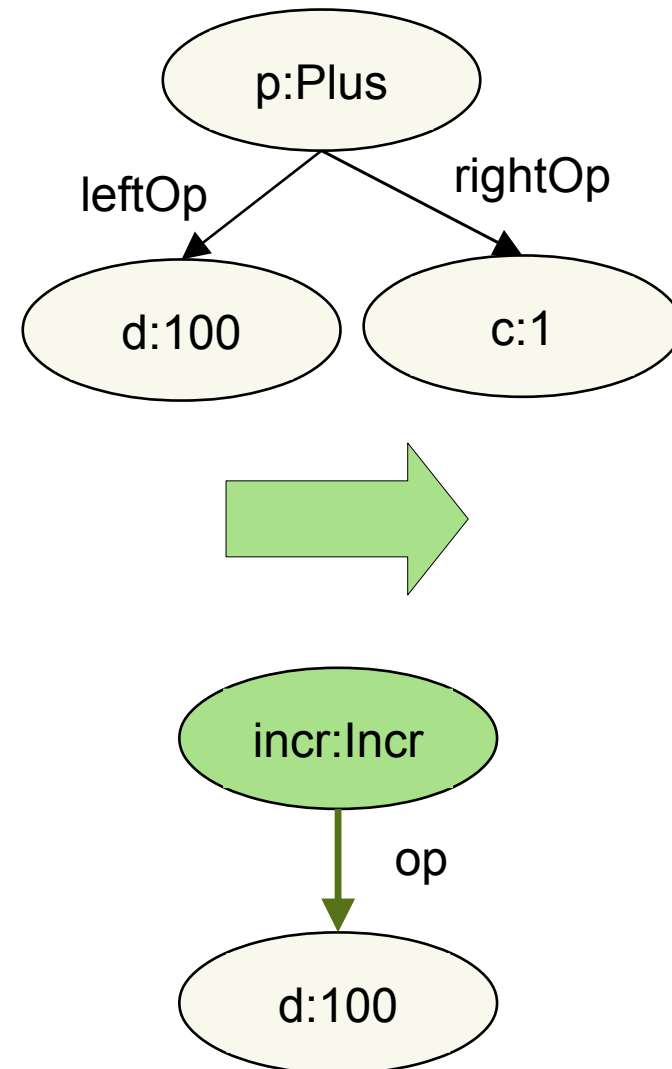
```
// if-then rules:  
if leftOp(p:Plus,v),  
    rightOp(p,c:1),  
then  
    Delete p,  
    Delete c,  
    Add incr:Incr,  
    op(incr,v);
```



```
FROM Plus [ leftOp [ var v ], rightOp [ c:1 ] ]  
CONSTRUCT incr:Incr [ var v ]
```

Context-Free Local Rewritings: Constant Folding

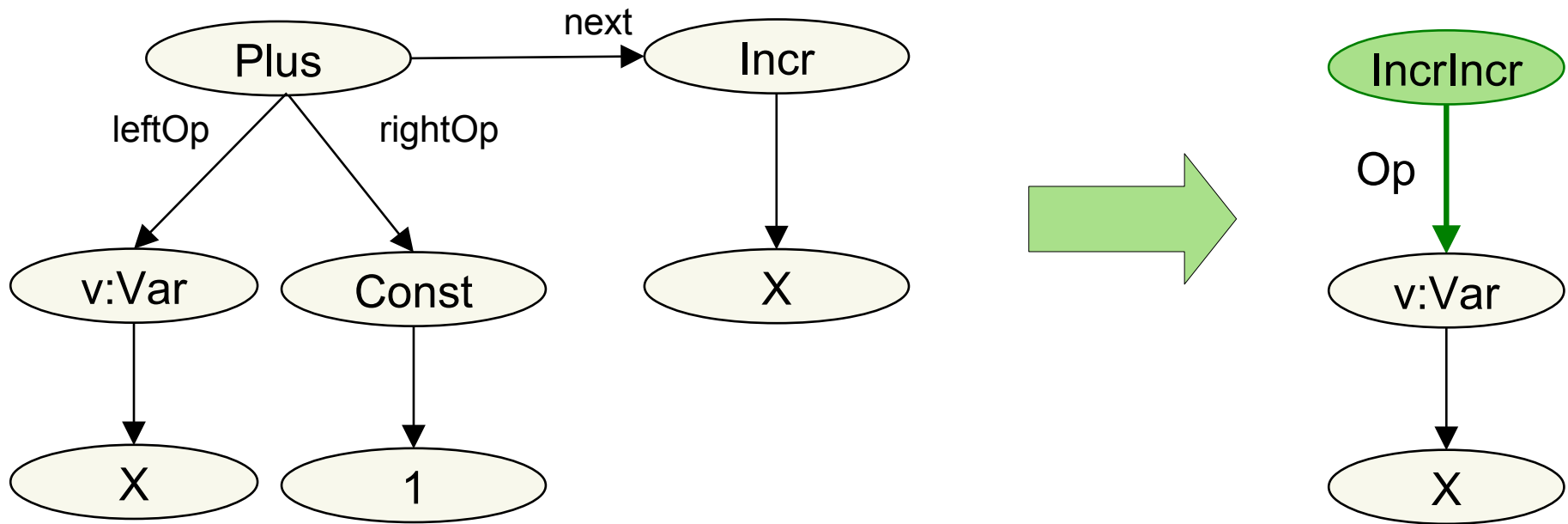
```
// if-then rules (logic):  
if leftOp(p:Plus,d:100),  
  rightOp(p,c:1),  
then  
  Delete p,  
  Delete c,  
  d.value=100,  
  op(incr,v);
```



```
FROM Plus [ leftOp [ var v ], rightOp [ c:1 ] ]  
CONSTRUCT incr:Incr [ var v ]
```

Peephole Optimization

- ▶ Peephole optimization is done on statement lists or trees
- ▶ Subtractive problem, because redexes are destroyed



```
FROM Plus[ leftOp[ var v:Var[X] ], rightOp[ Const[1] ],  
next[Incr[X]]  
CONSTRUCT IncrIncr[ Op[ var v ] ]
```


Rule Dependencies in a Set of Rules (here: All Superclasses as Transitive Closure in Inheritance Hierarchy): Simple Variant

```
CONSTRUCT // Base case of transitive closure
```

```
  subclassof-deriv [ var Sub, var Super ]
```

```
FROM
```

```
  subclassof [ var Sub, var Super ]
```

```
END
```

```
CONSTRUCT // Alternative, recursive rule
```

```
  subclassof-deriv [ var Sub, var Sup ]
```

```
FROM
```

```
  subclassof [ var Sub, var Z ],
```

```
  subclassof-deriv [ var Z, var Sup ]
```

```
END
```

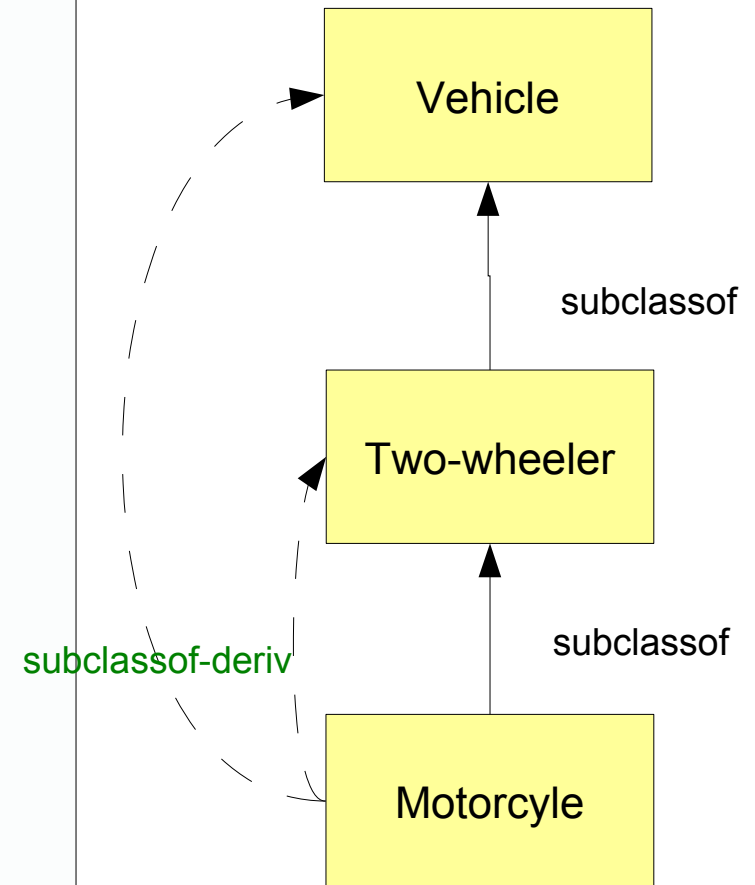
```
// Basic relation of direct subclassof
```

```
CONSTRUCT subclassof [ var Sub, var Sup ]
```

```
FROM
```

```
  in { resource { "file:...", "xml" },  
        <query> }
```

```
END
```



Rule Dependencies in a Set of Rules (here: All Superclasses as Transitive Closure in Inheritance Hierarchy) Variant with Self-Superclasses

```
CONSTRUCT // Base case of transitive closure
```

```
  subclassof-deriv [ var Cls, var Cls ]
```

```
FROM
```

```
  or { subclassof [ var Z, var Cls ],  
        subclassof [ var Cls, var Z ] }
```

```
END
```

```
CONSTRUCT // Alternative, recursive rule
```

```
  subclassof-deriv [ var Sub, var Sup ]
```

```
FROM
```

```
  or { // direct subclass  
        subclassof [ var Sub, var Sup ],  
        and {  
          subclassof [ var Sub, var Z ],  
          subclassof-deriv [ var Z, var Sup ]  
        } }  
END
```

```
END
```

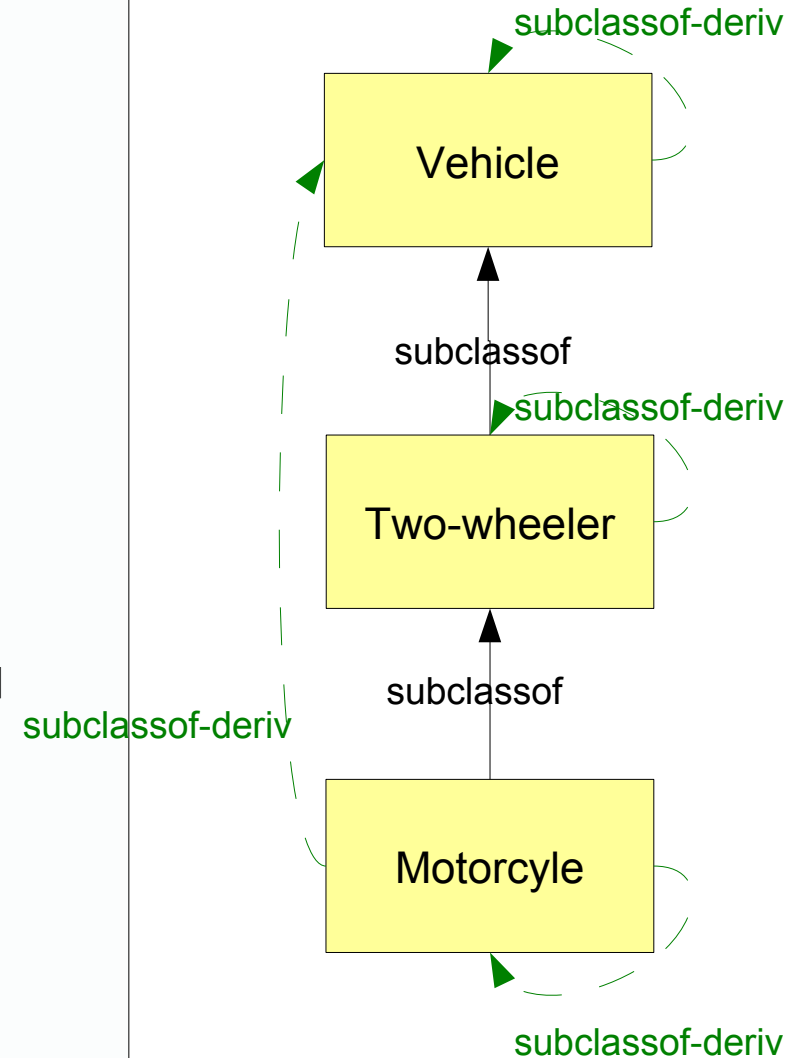
```
// Basic relation of direct subclassof
```

```
CONSTRUCT subclassof [ var Sub, var Sup ]
```

```
FROM
```

```
  in { resource { "file:...", "xml" },  
        <query> }
```

```
END
```



23.3.3 Larger Xcerpt Programs

- ▶ With modular Xcerpt



Modular Xcerpt

- ▶ *Modular Xcerpt* = Xcerpt + Module support
- ▶ http://www.reuseware.org/index.php/Screencast_LoadMXcerptProject_0.5.1
- ▶ Declaring a module in Modular Xcerpt:

```
MODULE module-id  
module-imports  
xcerpt-rules
```

- ▶ Declaring a module's interface
 - Modular Xcerpt programs importing a module can reuse public construct terms

```
public construct term
```

- Modular Xcerpt programs can provide data to an imported module's public query terms

```
public query term
```

- ▶ Modular Xcerpt is an Extension of Xcerpt for larger programs
- ▶ A query can be reused via a module's interface

IMPORT *module AS name*

- reuses public construct terms as a data provider for the given query term

in *module (query term)*

- provides the given construct term to public query terms of an imported module

to *module (construct term)*

Xcerpt Query Modules

- ▶ A *module* is a set of rules with interfaces
 - Interfaces define reusable query services
- ▶ Define a module: **MODULE** <name> <rule>*
- ▶ Define *input* and *output* interfaces:
public <query>; **public** <cterm>
- ▶ Import a module:
IMPORT <location> **AS** <name>
- ▶ Query module: **IN** <mod> (<query>)
- ▶ Provision module: **TO** <mod> (<cterm>)

Reusable module, in file:subclassof.mx

55 **IMPORT** file:subclassof.mx **AS** mod
Model-Driven Software Development in Technical Spaces (MOST)

```

GOAL vehicles [ all var Sub ]
FROM
  IN mod (
    output [[
      subclassof [ var Sub,
                  "Vehicle" ]
    ]] )
END

CONSTRUCT
  TO mod (
    input [
      subclassof [
        var Sub, var Sup ]
    ] )
FROM
  in { resource {
    "file:...", "xml" },
    <query> }
END
  
```

MODULE subclassof-reasoner

```

CONSTRUCT
  public output [
    all subclassof [ var Sub, var Sup ] ]
FROM subclassof-deriv [ var Sub, var Sup ]
END
  
```

```

CONSTRUCT
  subclassof-deriv [ var Cls, var Cls ]
FROM
  or { subclassof [ var Z, var Cls ],
    Subclassof [ var Cls, var Z ] }
END
  
```

```

CONSTRUCT
  subclassof-deriv [ var Sub, var Sup ]
FROM
  or { subclassof [ var Sub, var Sup ],
    and { subclassof [ var Sub, var Z ],
          subclassof-deriv [ var Z, var Sup ] }
  } }
END
  
```

```

CONSTRUCT subclassof [ var Sub, var Sub ]
FROM
  public input [[
    subclassof [ var Sub, var Sup ] ] ]
END
  
```

Result:

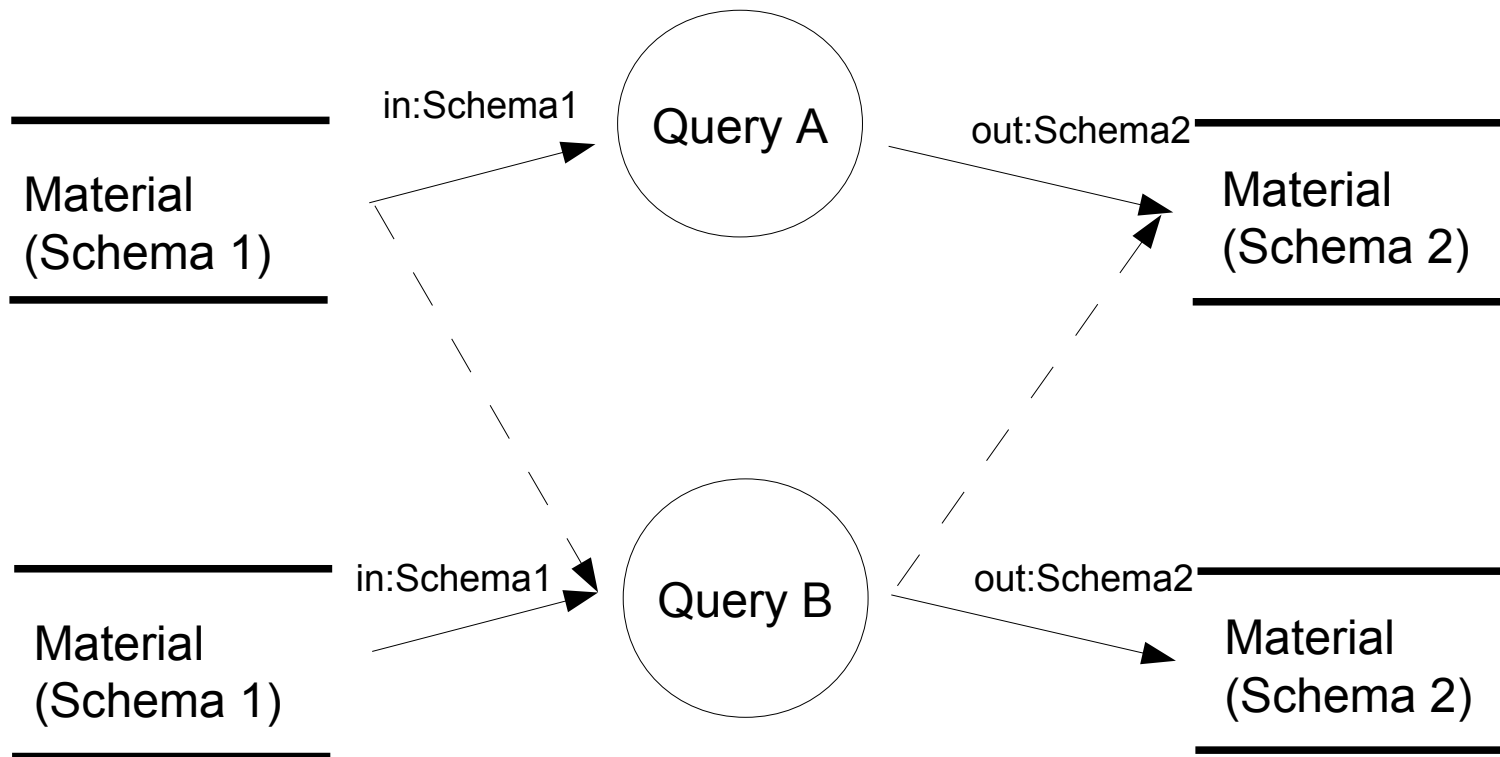
```

vehicles [
  "Vehicle", "Two-wheeler", "Motorcycle"
]
  
```

→ = data flow

Use of DQL and DTL in Werkzeugen

- ▶ Stream-processing QL and TL are useful for the composition of tools on material streams



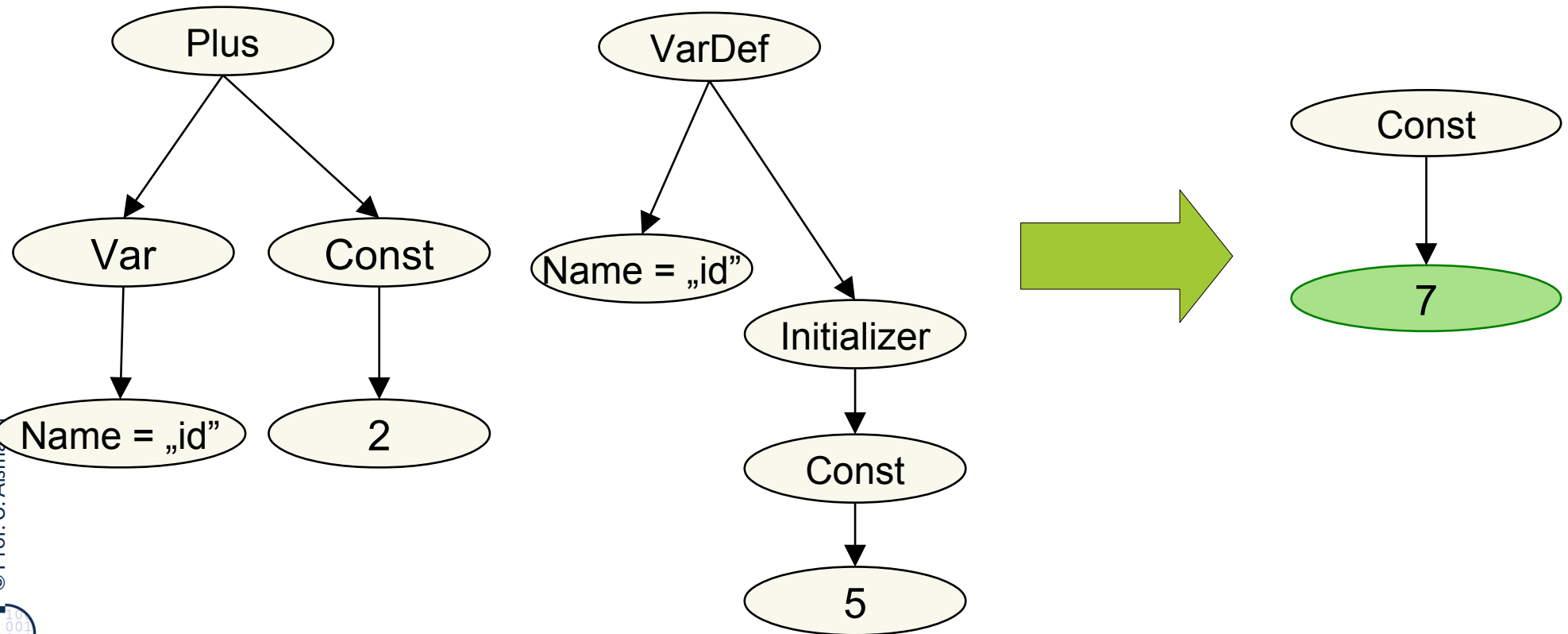
Tool composition theorem 1: Two tools can be composed if their input and output types are compatible and the output order in the output streams is commutative

23.3.4 Context-Sensitive Term Rewritings



Extended Constant Folding as Subtractive TRS

- ▶ A term rewrite system usually works context-free, i.e., matches and rewrites only one term.
- ▶ A **context-sensitive term rewriting** matches a set of non-connected left-hand side terms with a redex.
 - Matching of one redex can be done in quadratic time, because non-connected nodes have to be pairwise compared



Covered Code Optimizations

59

Model-Driven Software Development in Technical Spaces (MOST)

▶ Global transformations

- Refactorings:
 - **Rename** all uses of a variable and its definition
 - **Move** a method into another class
 - **Split** a class
- Code motion
 - Move an expression out of a loop
 - **Clone removal:** Outline a method from a set of clones and transform all clones to calls of the method

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

- ▶ Interpretation and abstract interpretation on syntax link-trees

Rep.: Attribute Grammars (AG)

- ▶ An **attribute grammar** describes an interpreter on a syntax tree (a hierarchical program representation)
 - The syntax tree is described by a 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)** which define interpretation functions on the syntax tree
 - Usually, the rules are interpreted with recursion along the attributed syntax tree
- ▶ *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

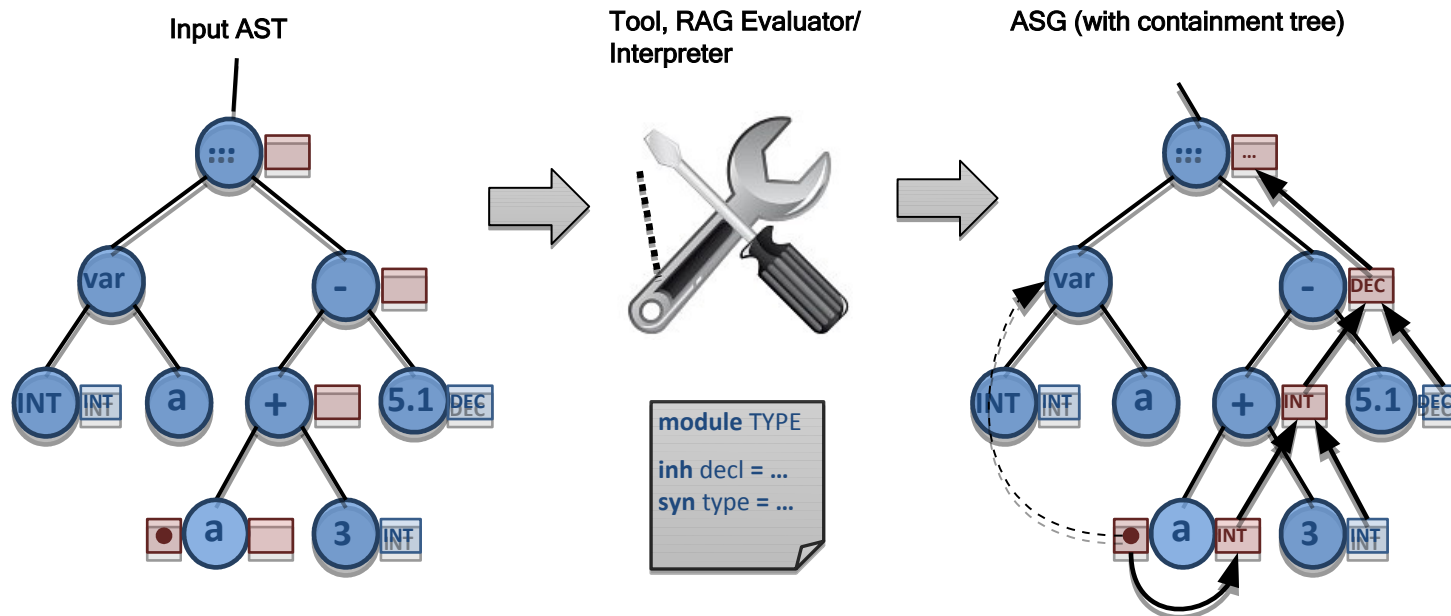
Reference Attribute Grammars (RAG) Work on Link Trees

- ▶ 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)
 - Graph representations *are* possible in pure AGs
 - 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

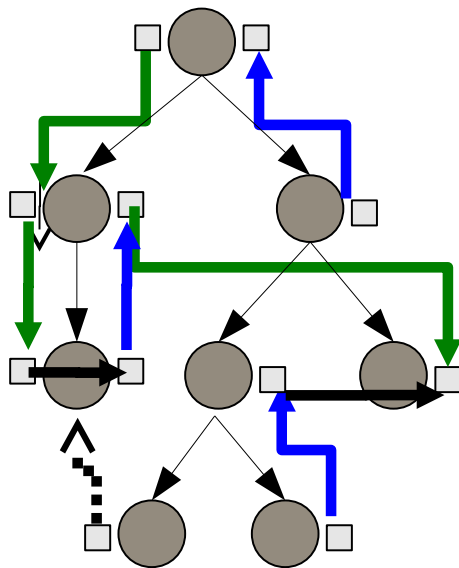
RAG-based abstract interpreters can analyse, interpret, and abstractly interpret models

What is a Reference Attribute Grammar (RAG)?

- ▶ **Attributes compute static semantics over syntax trees** [Knuth68]
 - Basis: (context-free) grammars + attributes + semantic functions
- ▶ **Attribute types:**
 - **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 existing nodes in the AST
- ▶ **Tool:** www.jastadd.org



Kinds of Attributes and Attribute Dependencies

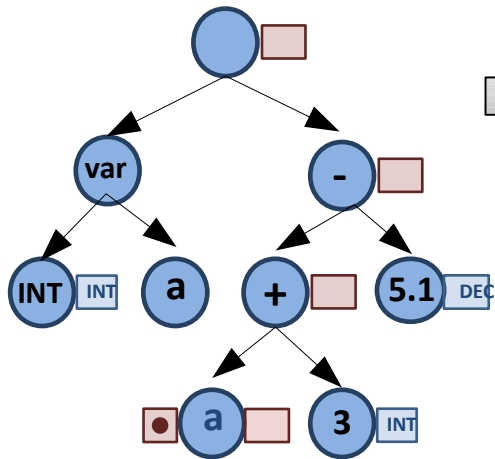


- **Inherited attributes** (inh, green): Top-down value dataflow/computation
- **Synthesized attributes** (syn, blue): Bottom-up value dataflow/computation
- **Collection attributes** (coll): Collect values freely distributed over the AST
- **Reference attributes** (dashed): Compute references to existing nodes in the AST

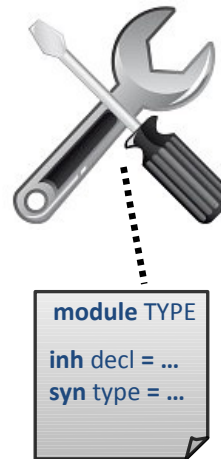
Basic Working Principle of RAG Tools

- ▶ Computing attribution functions
- ▶ Setting references (dashed edge)

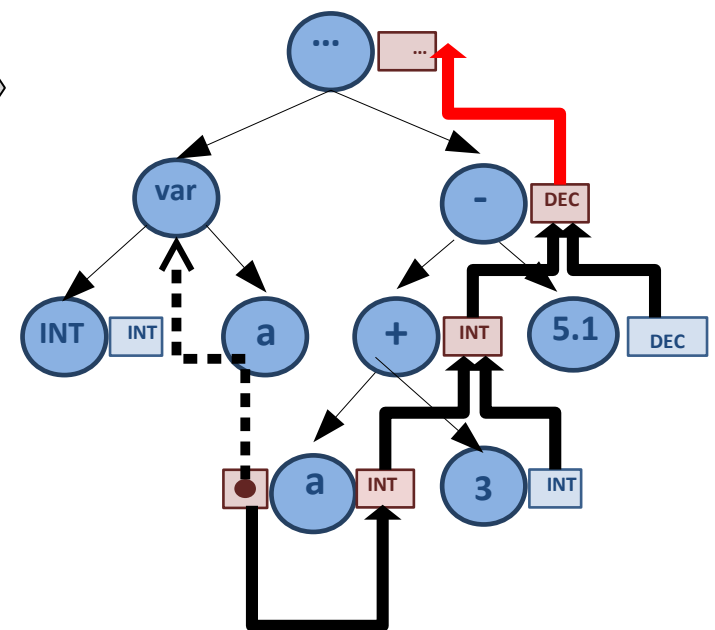
Input AST (from parser/
editor/transformer)



Tool: RAG Evaluator
(generated or interpreted)



AST with overlay graph

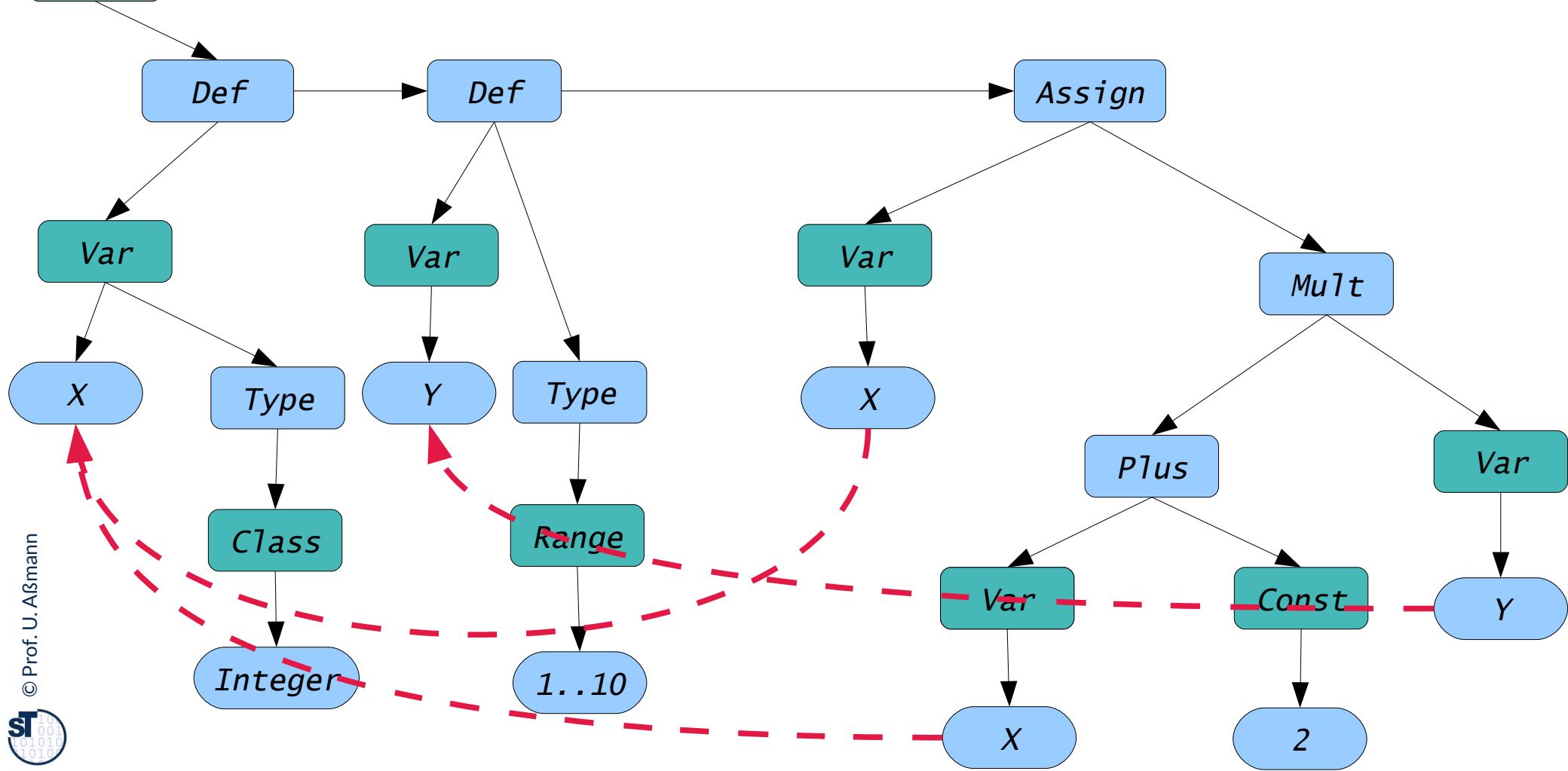


Why Links? (1) Name Analysis in Programs

- ▶ Name analysis searches the right definition for a use of a variable and materializes it as cross-tree link

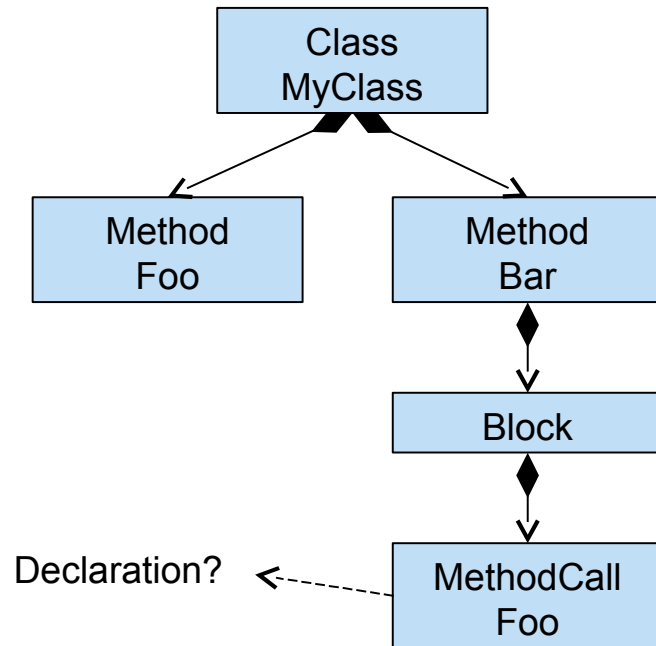
This holds for models and programs in *any* language

Proc



Why Links? (1) Name Analysis for Calls

- ▶ Name 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



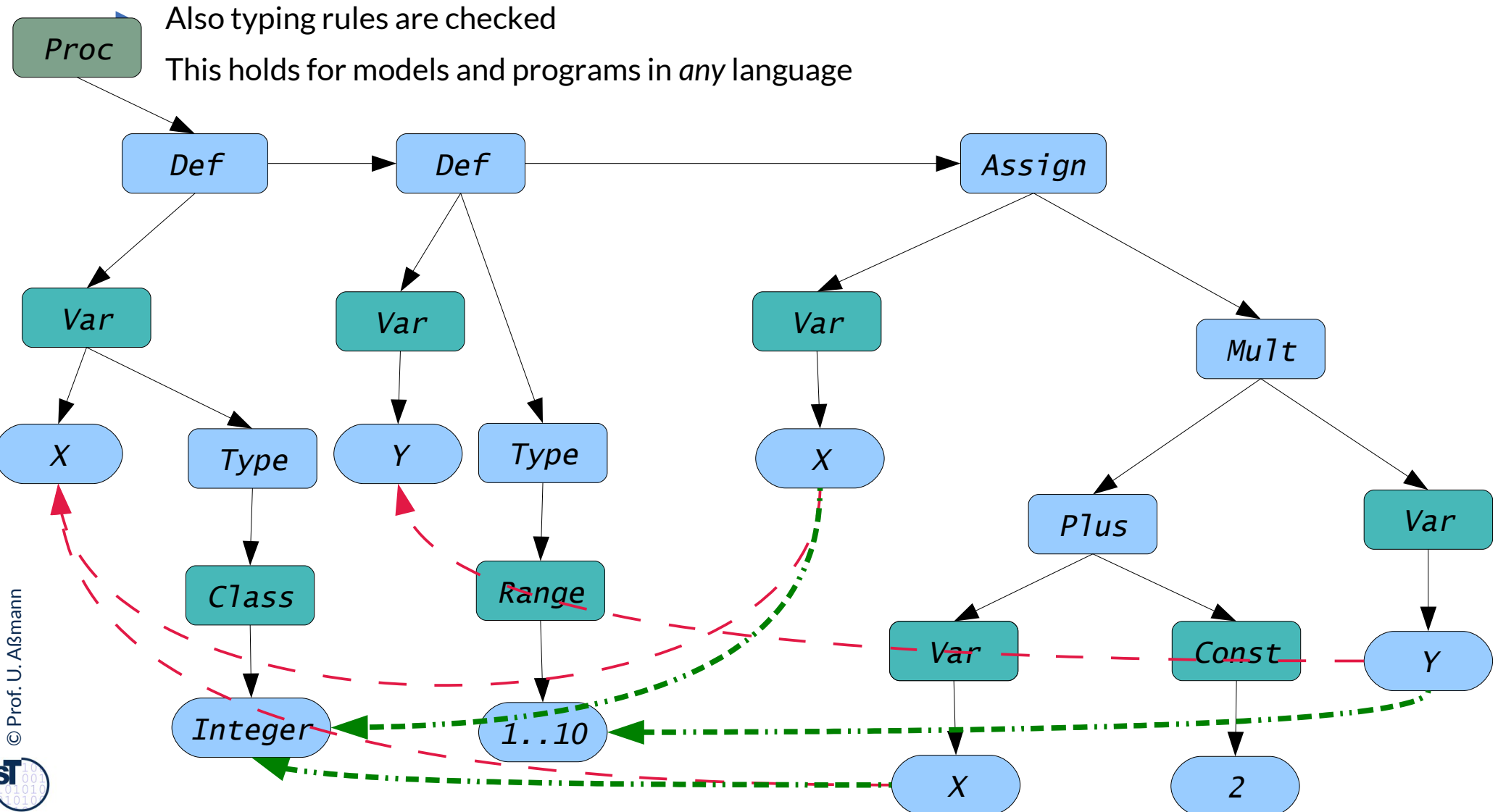
Problem:
Many possible name resolutions (Shadowing, overloading, several namespaces, namespace modifiers e.g. super, etc.)

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

Also typing rules are checked

This holds for models and programs in any language

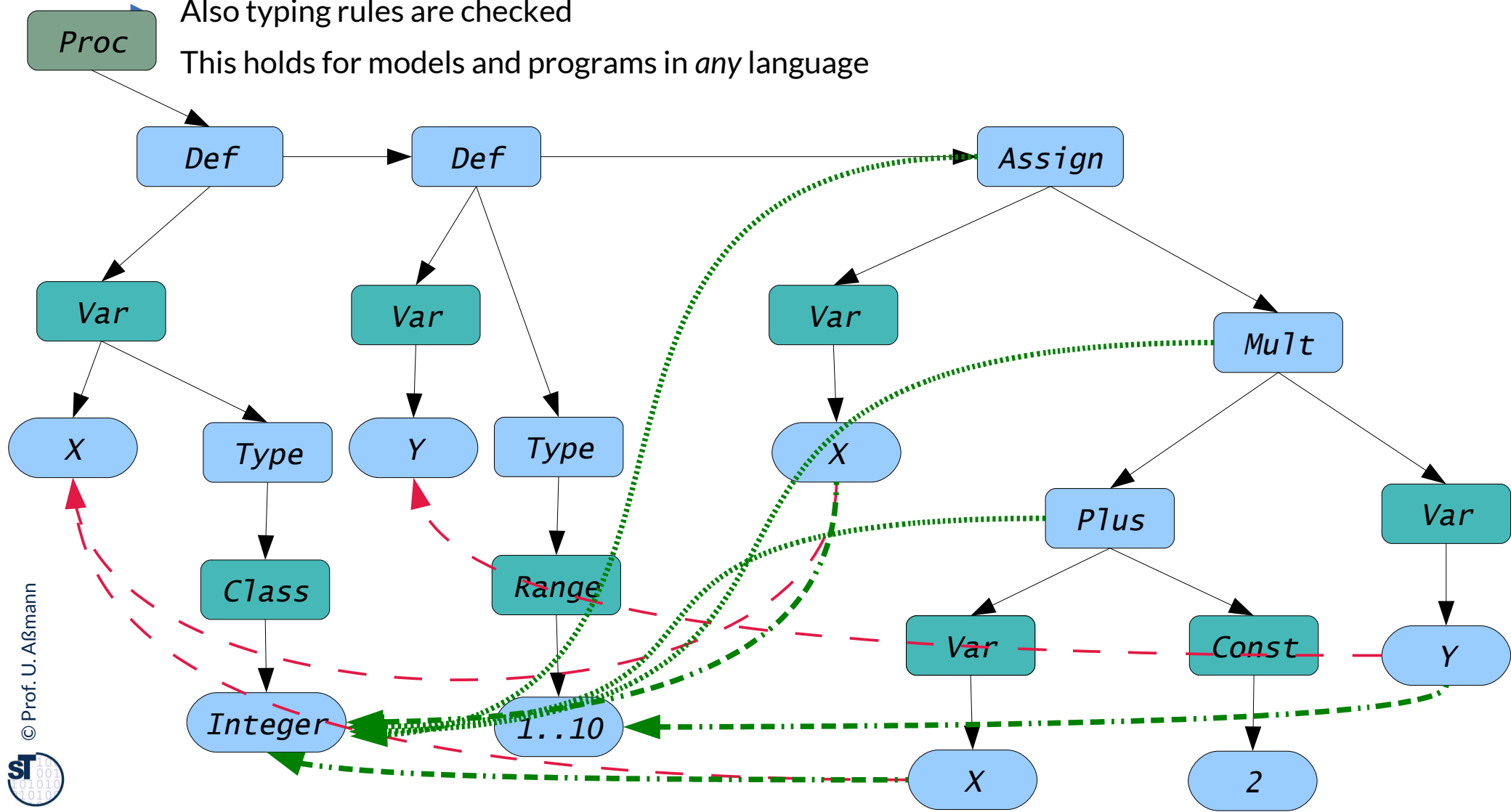


Why Links? (2) Type Analysis in Programs

- ▶ 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

Also typing rules are checked

This holds for models and programs in *any* language



(Beaver) Grammar of SiPLE, a Simple Prog. Lang.

<https://bitbucket.org/jastemf/jastemf-plugins/src/4290860b492fcd10ac645b02eae64643cedf8192/jastemf-examples/siple/org.jastemf.siple/speci>

70

Model-Driven Software Development in Technical Spaces (MOST)

```
CompilationUnit = DeclarationList.decls
    {: return new Symbol(new CompilationUnit(decls)); :}
;

DeclarationList = Declaration.decl
    {: return new Symbol(new List<Declaration>().add(decl)); :}
;
| DeclarationList.list Declaration.decl
    {: list.add(decl); return _symbol_list; :}
;

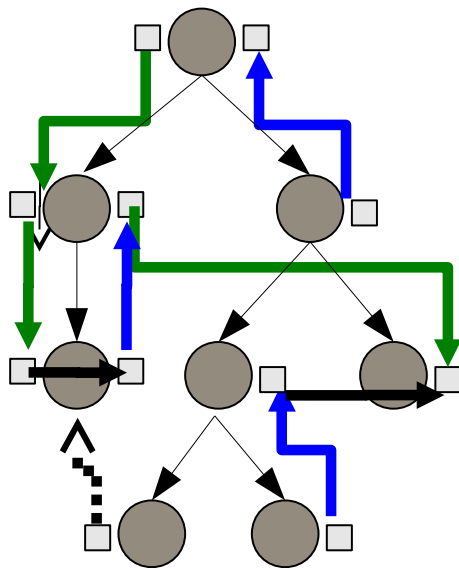
Declaration = VariableDeclaration.decl pSEMICOLON
    {: return _symbol_decl; :}
| ProcedureDeclaration.decl pSEMICOLON
    {: return _symbol_decl; :} ;

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; :}
;
```

Kinds of Attributes and Attribute Dependencies



- **Inherited attributes** (inh, green): Top-down value dataflow/computation
- **Synthesized attributes** (syn, blue): Bottom-up value dataflow/computation
- **Collection attributes** (coll): Collect values freely distributed over the AST
- **Reference attributes** (dashed): Compute references to existing nodes in the AST

JastAdd is an Object-oriented ReCRAG evaluator generator

- Generated evaluators are demand-driven
- Handles combination of semantics, evaluation order and tree traversal

Two specification languages (AST 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
- For each attribute equation a method implementation is generated

The generated class hierarchy is the attribute evaluator.

The JastAdd Approach

JastAdd: AST and Attribute Specifications

AST specification example:

```
abstract Stmt;  
If:Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];  
abstract Decl:Stmt ::= <Name:String>;  
ProcDecl:Decl ::= Para:VarDecl* Body:Block;  
VarDecl:Decl ::= <Type>;
```

Attribution example in JastAdd:

```
syn Type Expr.Type(); // Type: Enumeration class of all types  
eq BinExpr.Type() = ...; // Default equation  
eq Equal:BinExpr = ...; // Refined equation  
inh Block Stmt.CurrentBlock(); // Reference attribute  
eq Block.getStmt(int index).CurrentBlock() = this;
```

Template-SiPLE Grammar (in RTG Notation of JastAdd)

```
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;
```

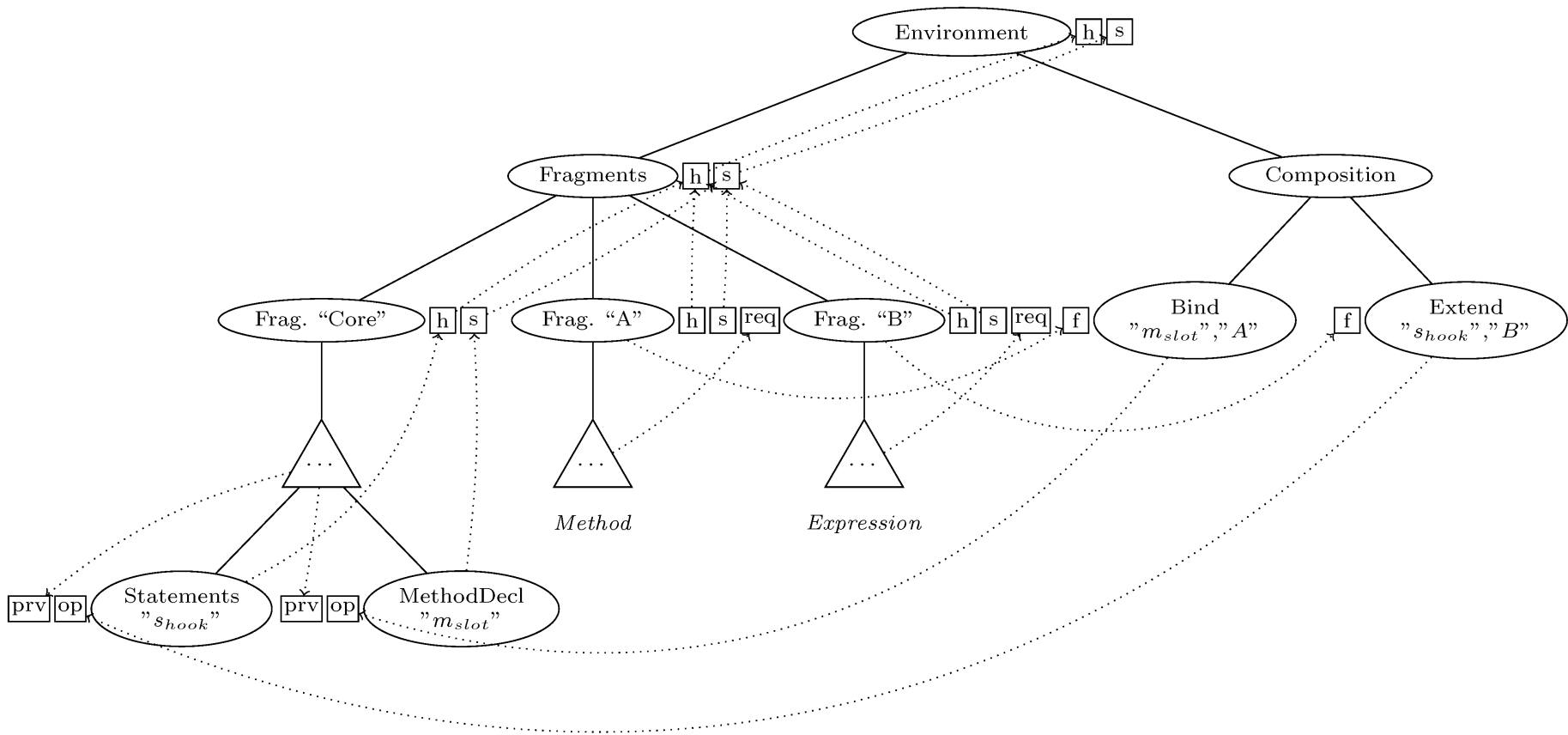
```
UMinus:UnaryExpression;
```

```
abstract BinaryExpression:Expression ::=
```

```
Operand1:Expression
```

```
Operand2:Expression;
```

RAG over Template-SIPLE



Summary

- ▶ Compiler-Frontends for Textual Languages can be produced with JastAdd (RAG)
- ▶ After parsing, the RAG processes links for the pure tree
 - Completing the link tree with references
 - Name analysis, type analysis, wellformedness constraints

23.5 Reference Attribute Grammars for Interpreters and Analyzers on Syntax Link-Trees of Models

- ▶ Interpretation and abstract interpretation on syntax link-trees with the tool JastEMF
- ▶ [Www.jastEMF.org](http://www.jastEMF.org)

23.2.2 The JastEMF Approach for Static Analysis of Models

Metamodelling Languages, Tree Structures and AGs



Claim:

Most metamodeling languages' metamodels separate model instances into

- A tree structure (AST) and
- A graph structure based on references between tree nodes (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 ASGs based on ASTs and to reason about ASGs

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

Each model instance of an Ecore metamodel has a spanning tree

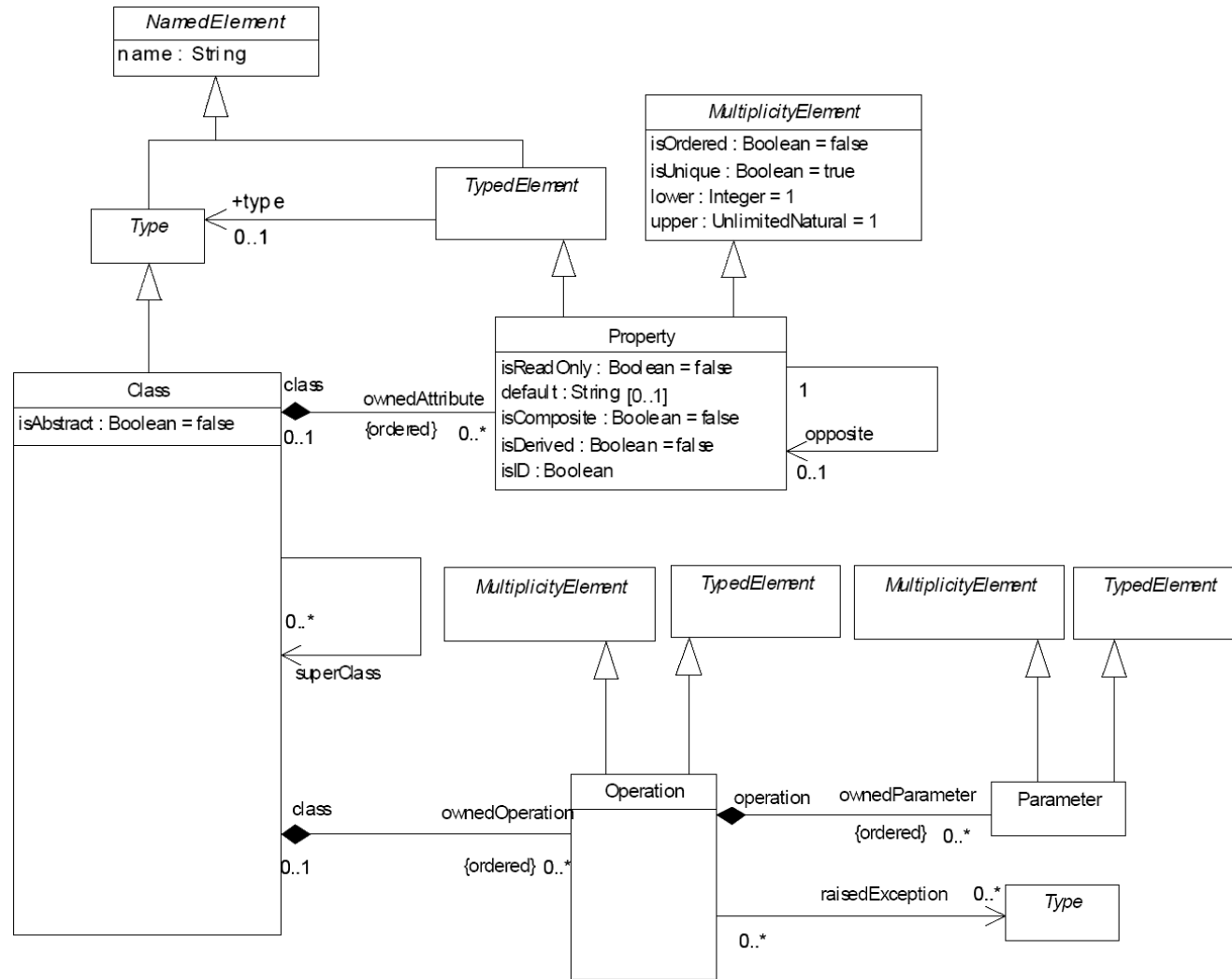
- Its set of nodes are all metaclass instances (Non-terminals) and non-derived properties (Terminals)
- Its edges are metaclass instances' containment references

Model instances' semantics are

- Derived properties (ASG)
- Non-containment references (ASG)
- Operations

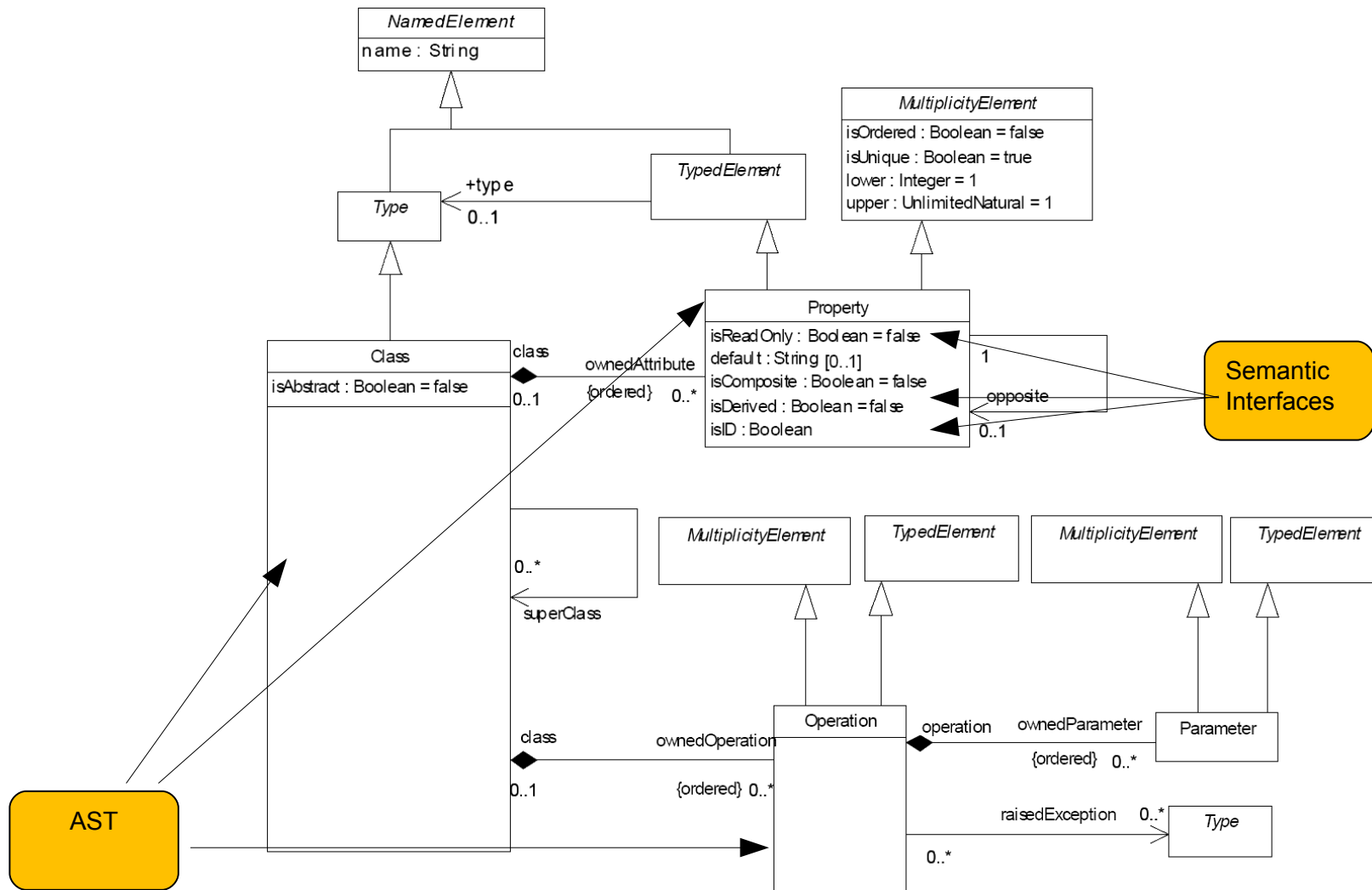
Derived properties and non-containment references = ASG on top of the spanning tree.

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

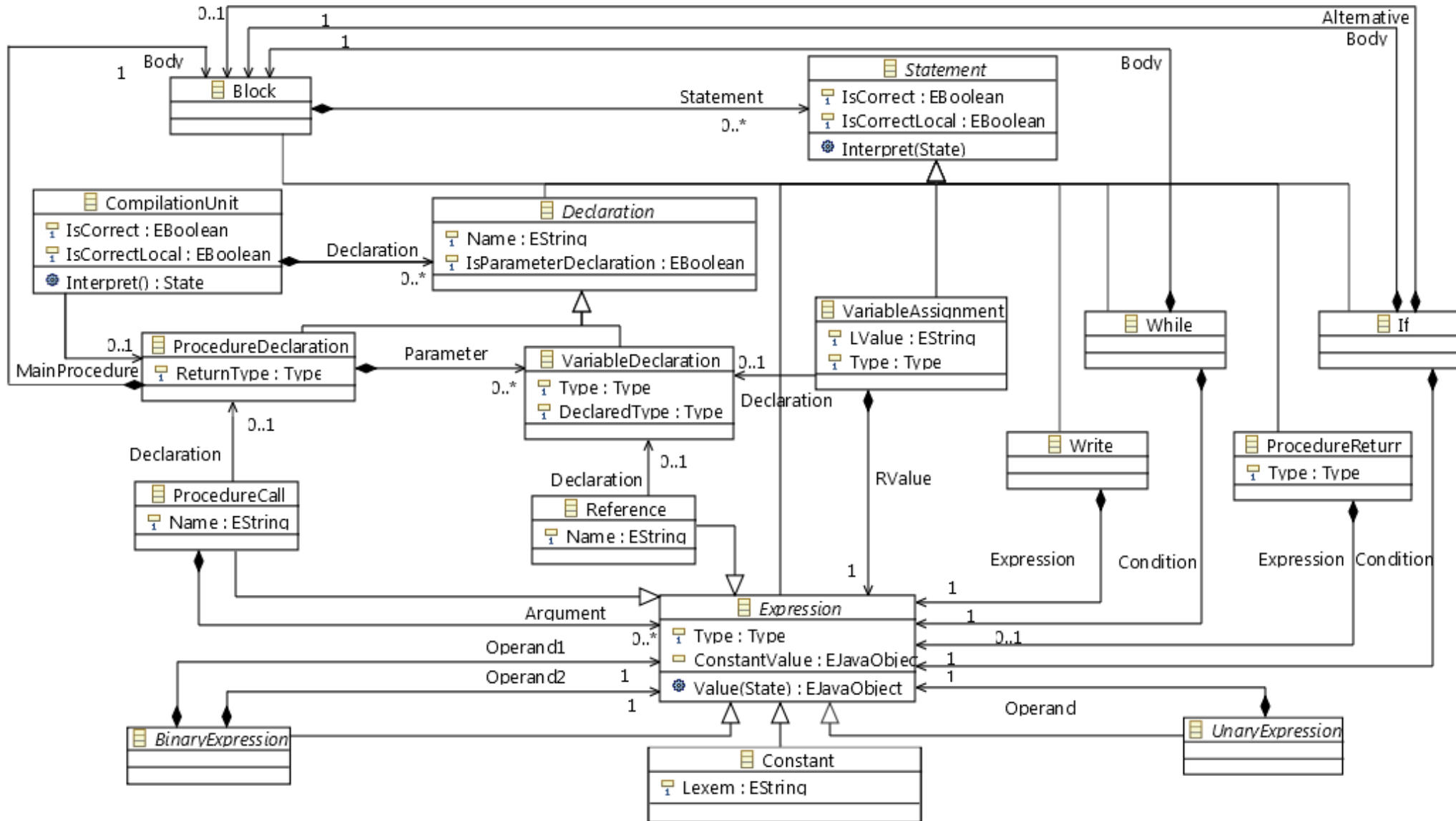


Where is the spanning tree?

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

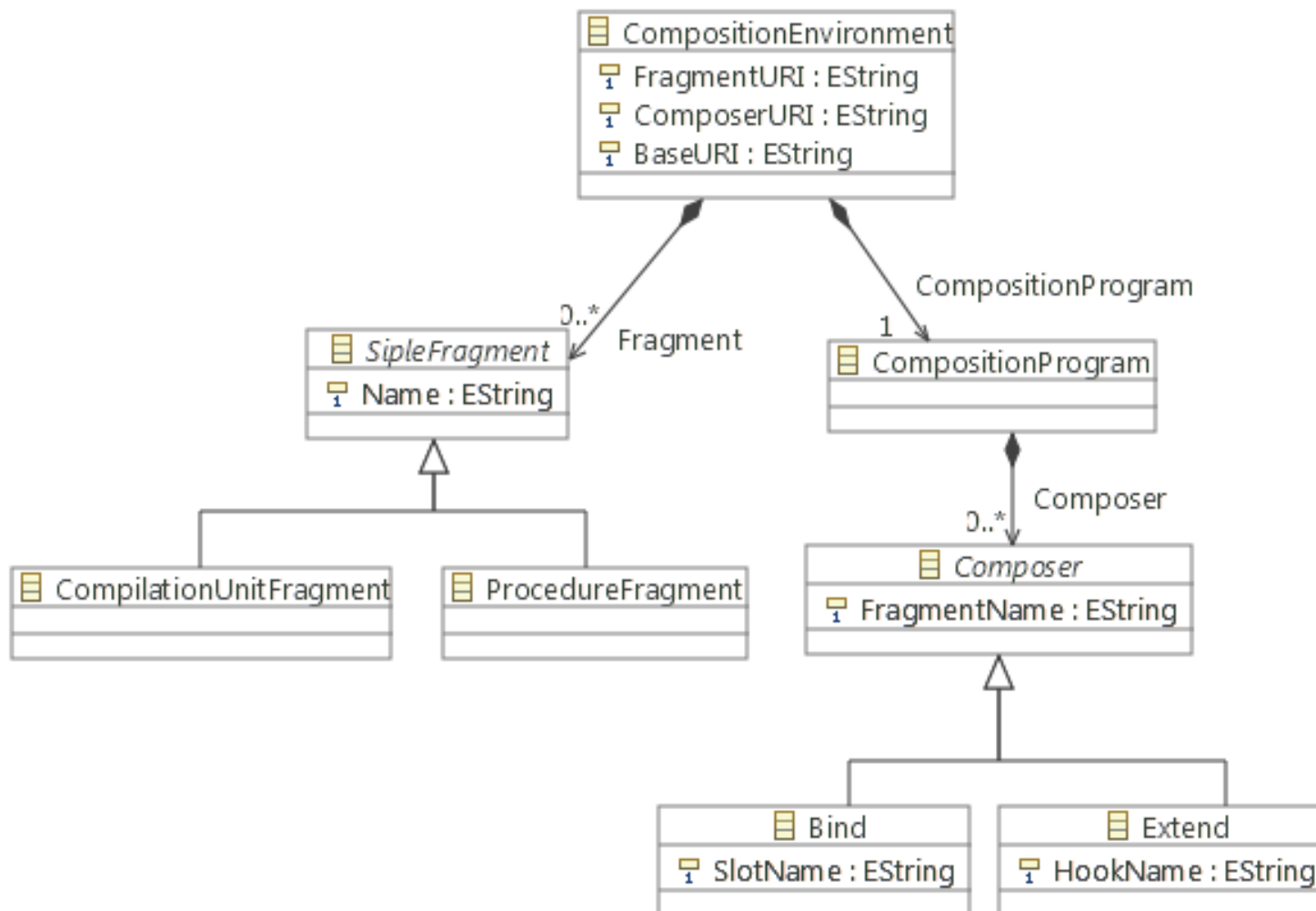


SiPLE EMF Metamodel



Where is the spanning tree?

Template-SiPLE EMF Metamodel (for the AST)



Where is the spanning tree?

The JastEMF Approach Requires a Ecore-JastAdd Concept Mapping

In summary: EMF and JastAdd generate a class hierarchy

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

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

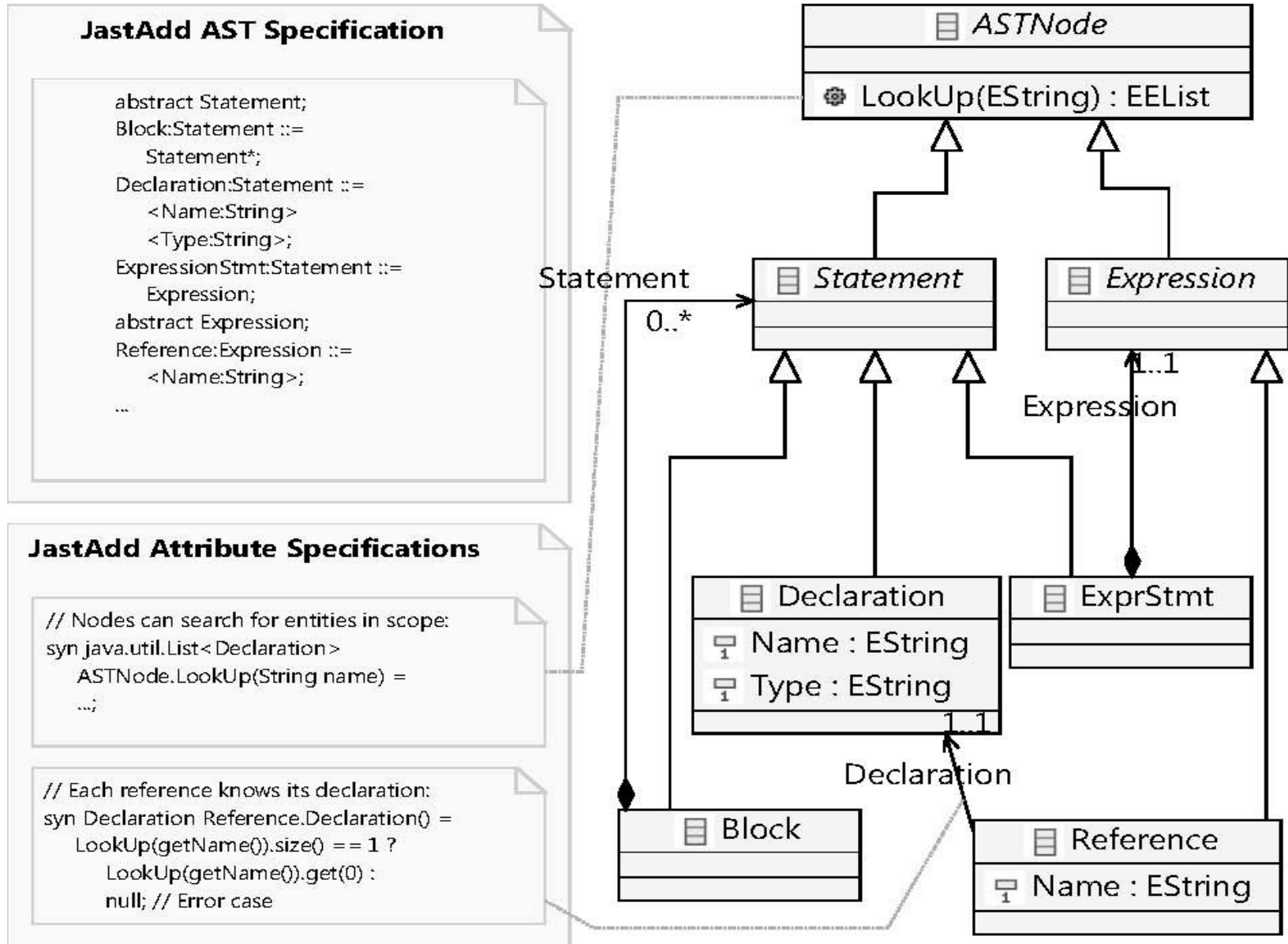
- For every derived property: JastAdd attribute of equal name and type
- For every non-containment reference: JastAdd reference attribute of equal name and type
- For side effect free operations: JastAdd attribute of equal signature
- Metamodel AST (Metaclasses; non-derived properties; containment references)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

<https://bitbucket.org/jastemf/jastemf-plugins/wiki/Approach.md>

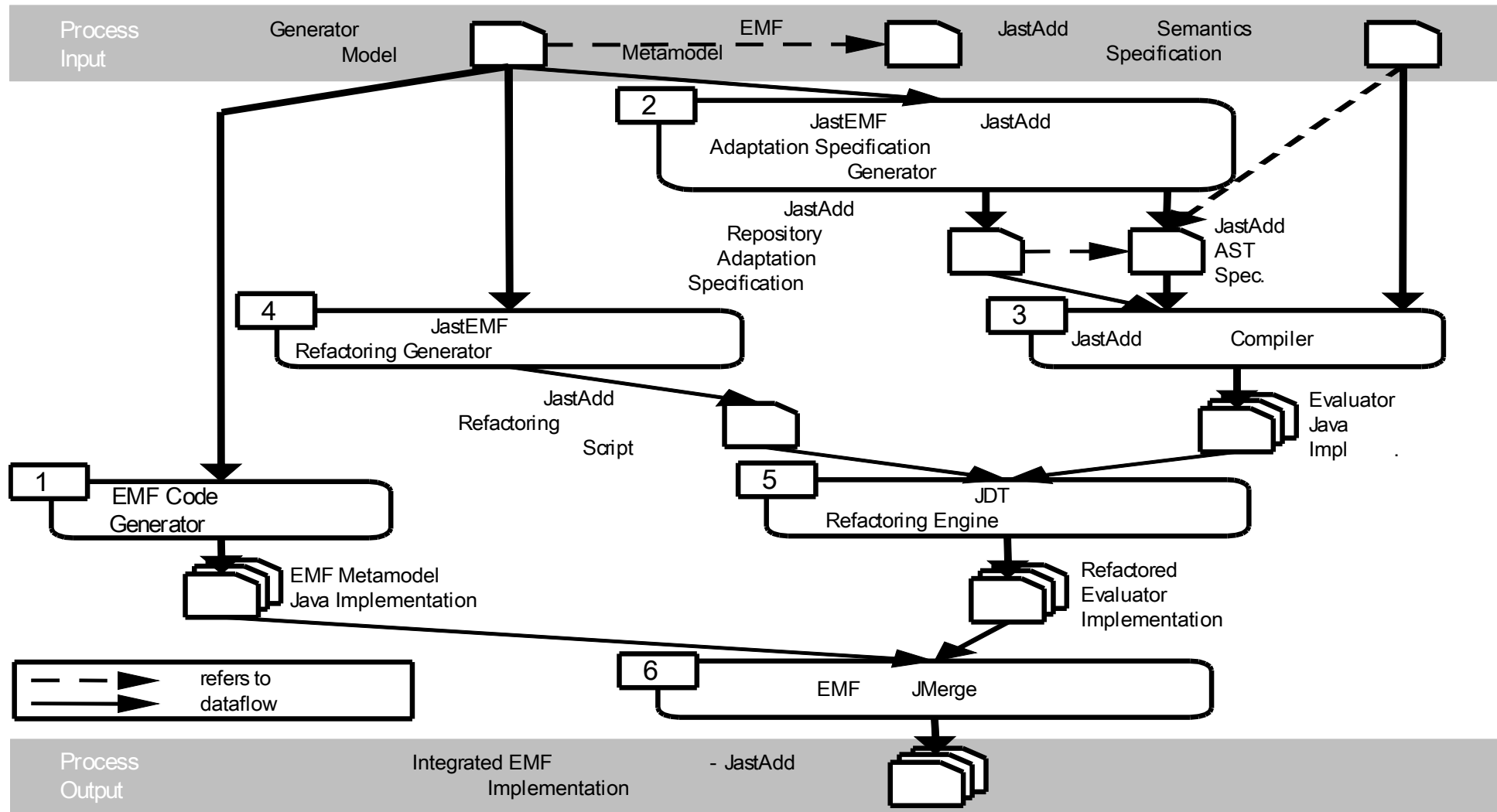
The JastEMF Approach Requires a Ecore-JastAdd Concept Mapping

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 Concept Mapping

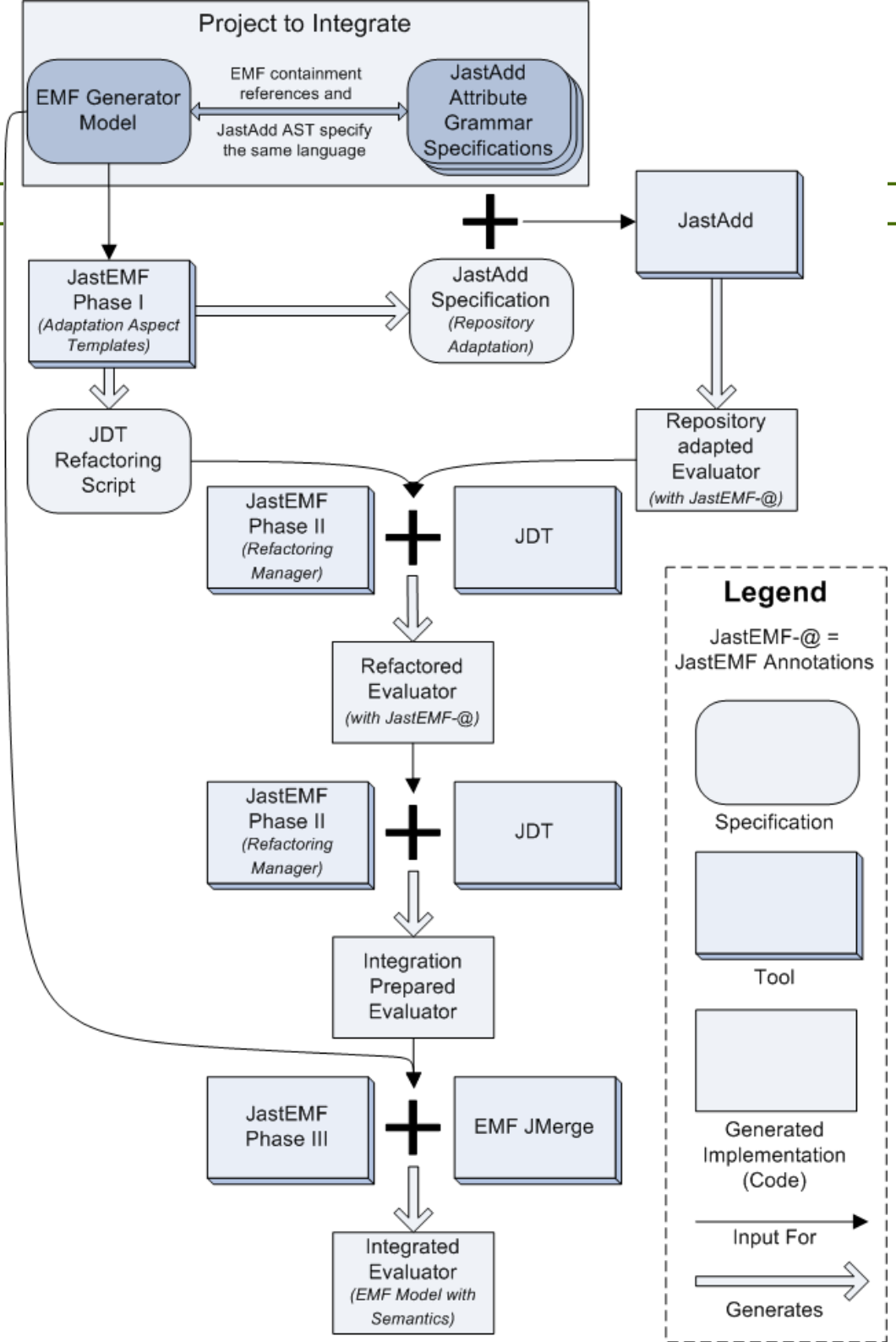


JastEMF's Integration Process of EMF and JastAdd



JastEMF steers EMF & JastAdd
 EMF and JastAdd development can be handled as usual

JastEMF's Integration Process of EMF and JastAdd



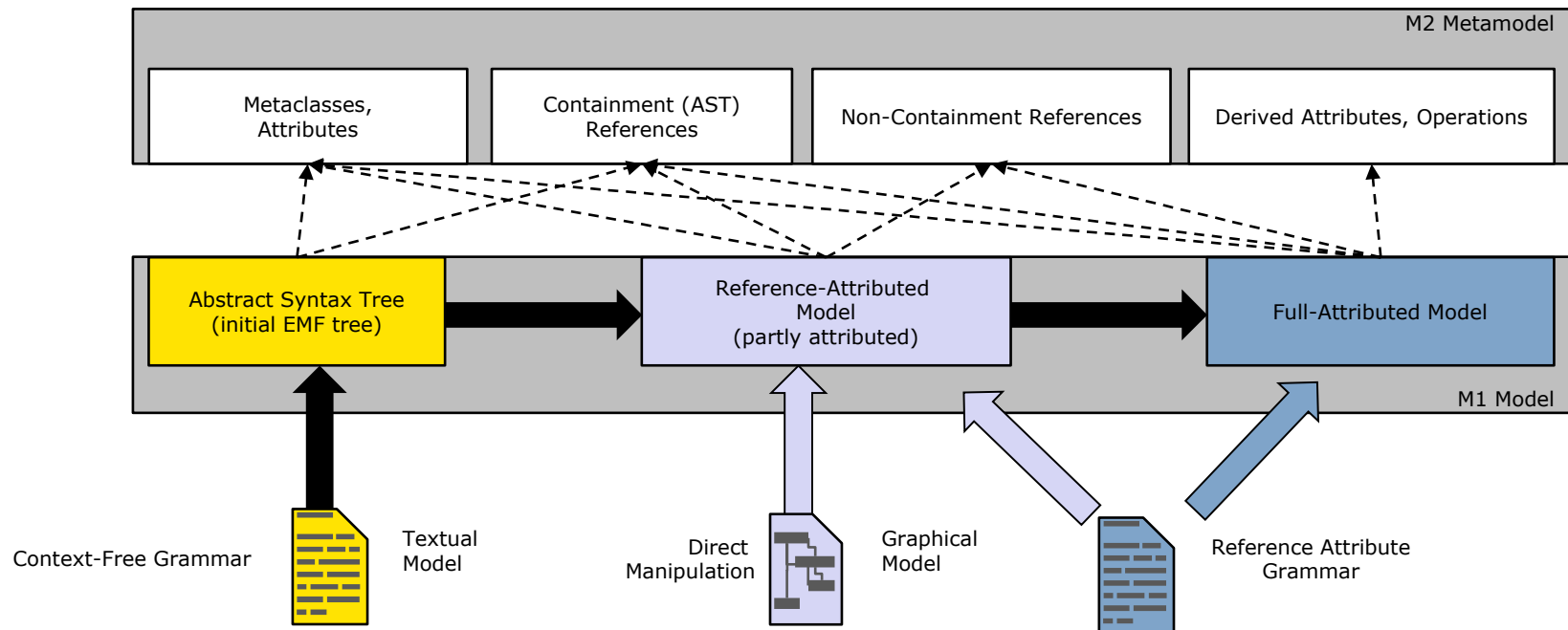
JastEMF steers EMF & JastAdd
EMF and JastAdd development
can be handled as usual

<https://bitbucket.org/jastemf/jastemf-plugins/wiki/Approach.md>

Stepwise Attribution of Link-Trees

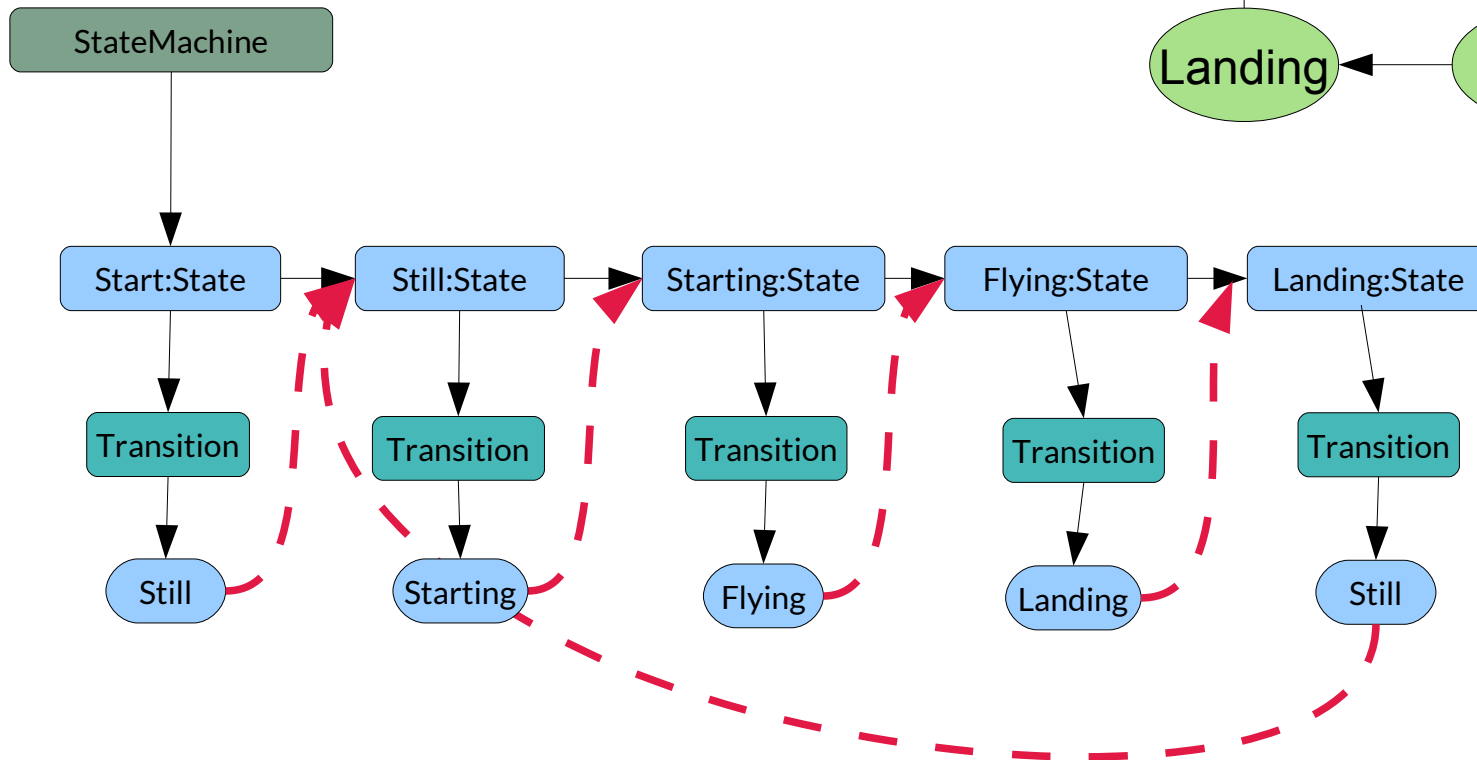
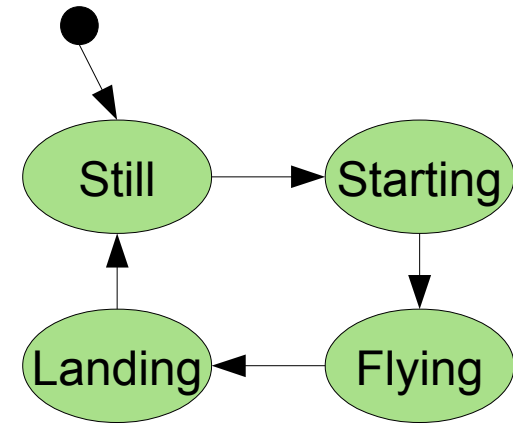
Semantic evaluation can start from (partly) reference-attributed EMOF models

- 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



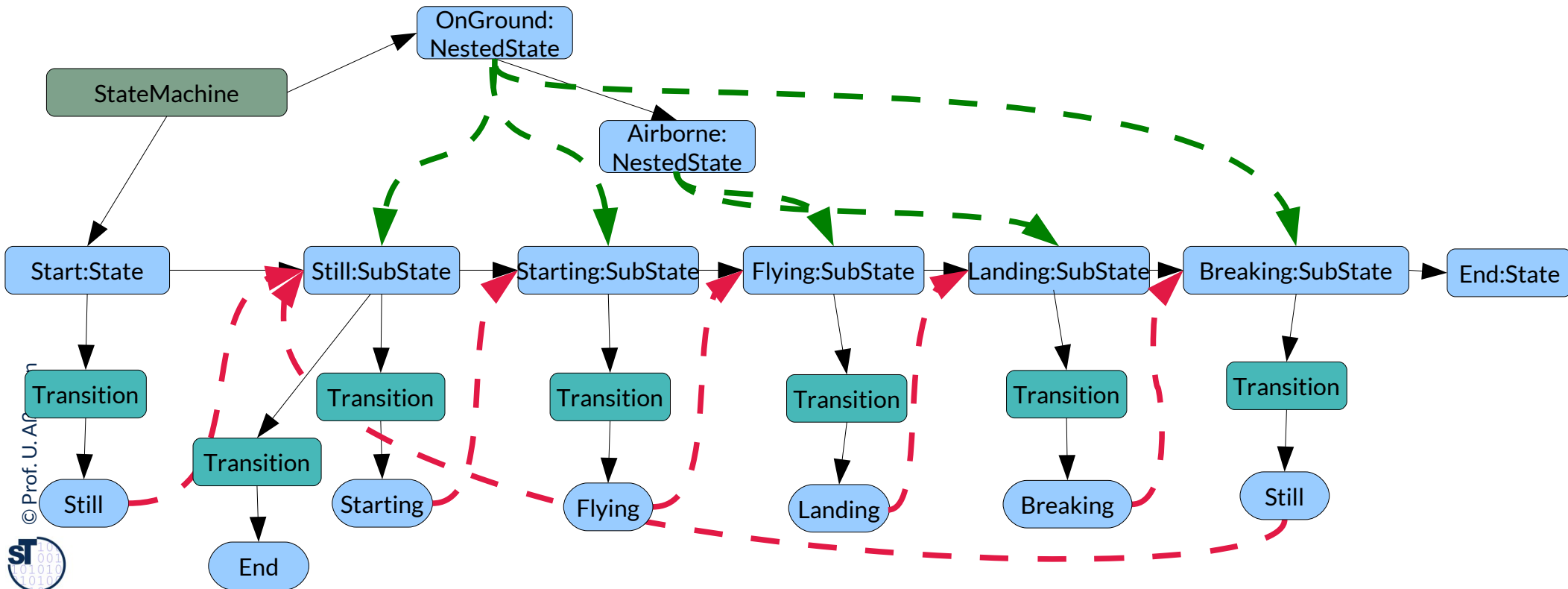
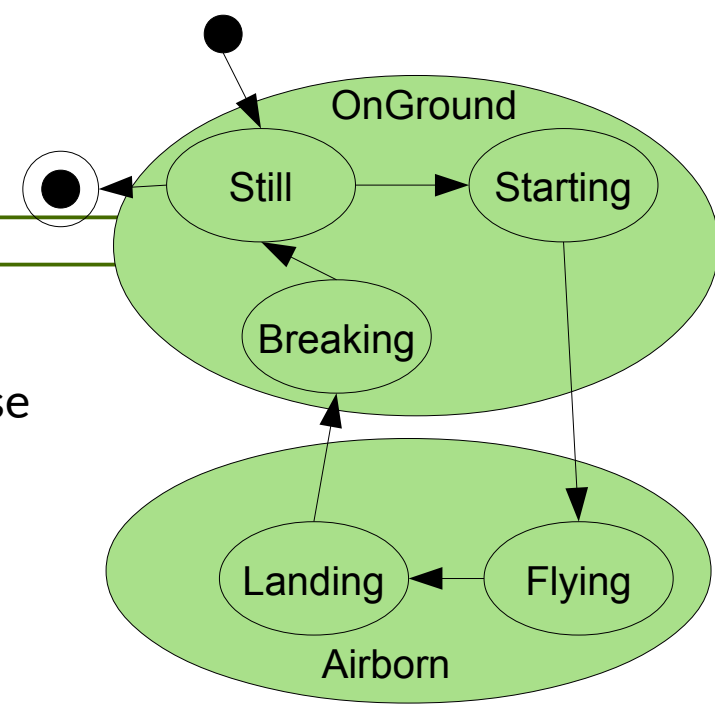
Why Links? (2) Name Analysis in Models

- ▶ 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

- ▶ 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 s:NestedState: s has-link-to [1..n] Substate AND s NOT has-subtree

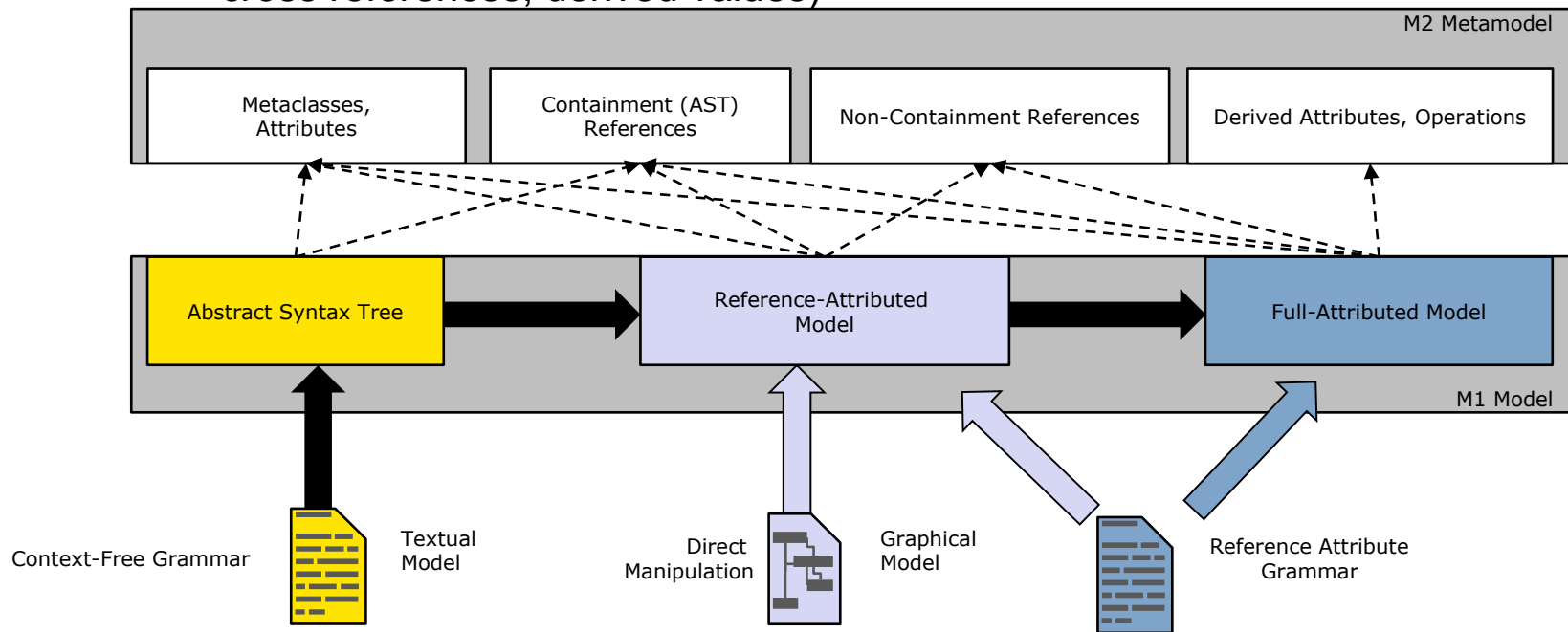


EMOF and Reference Attribute Grammars

- ▶ Ecore (EMOF) models are ASTs with 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

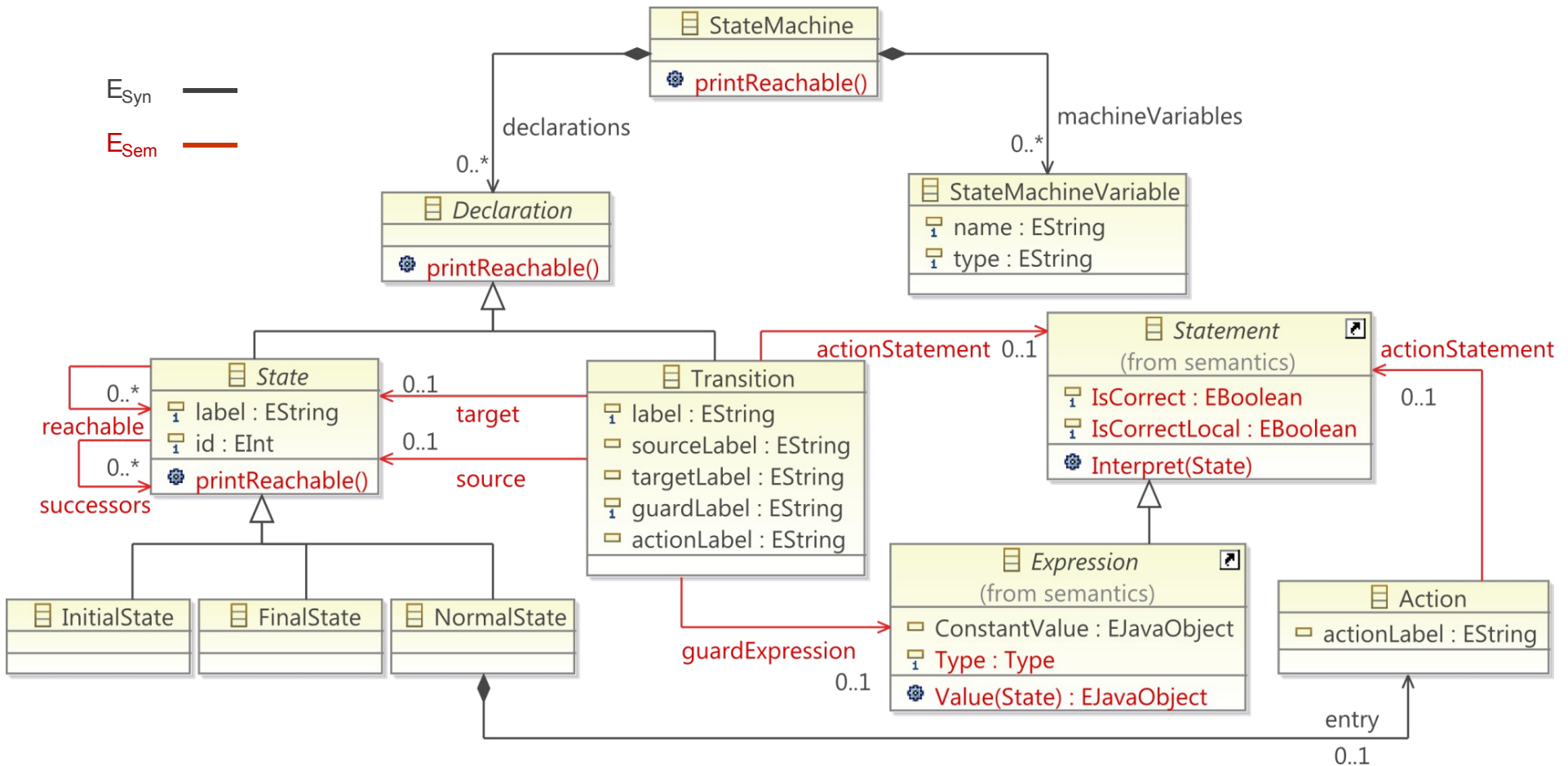
- ▶ EMOF models are ASTs with cross-references and derived information!
- ▶ **Tool:** www.jastemf.org

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

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

Example 1: Statechart Metamodel in EMOF

Where is the spanning tree?



(Ecore-based, extended version of Statechart example in Hedin, G.: Generating Language Tools with JastAdd. In: GTTSE '09. LNCS, Springer (2010), 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
```

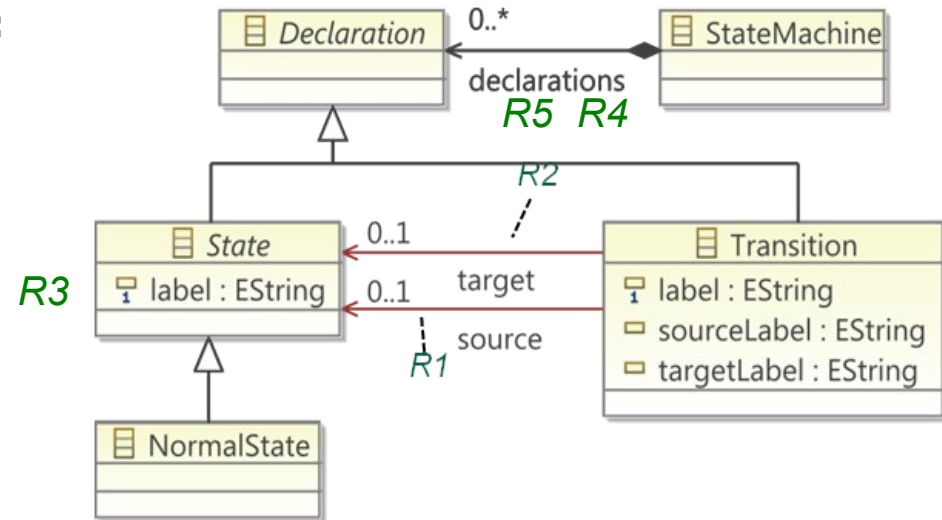
```
syn State Declaration.localLookup(String label) = (label==getLabel()) ? this : null; // R5
```

// inherited functions (top-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 Statechart example in Hedin, G.: Generating Language Tools with JastAdd. In: GTTSE '09. LNCS, Springer (2010), see also www.jastemf.org)

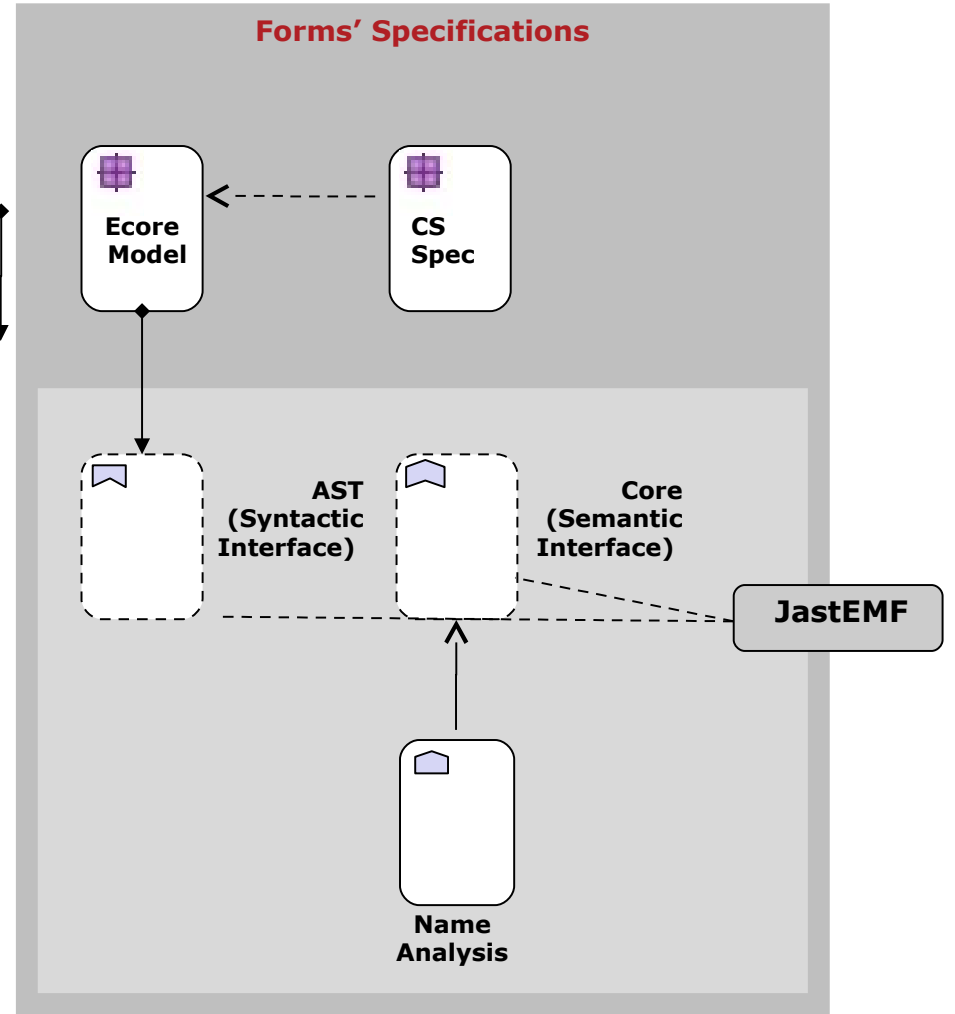
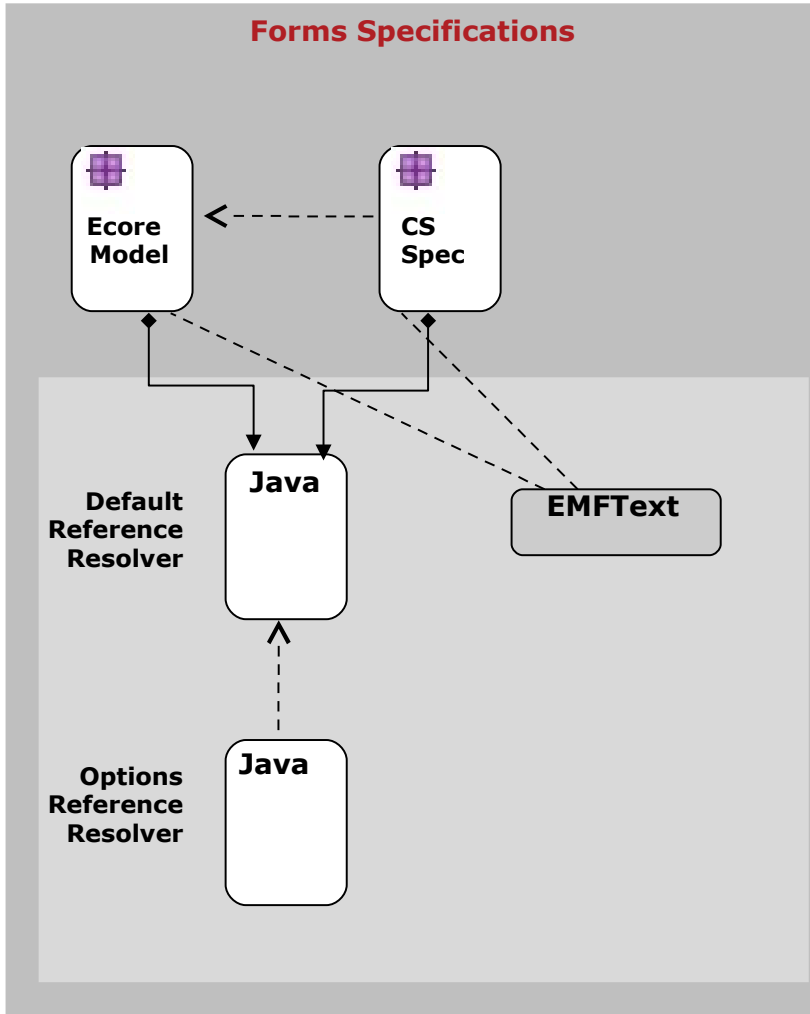
Example 1: Generated Statechart Editor with Runtime Semantic Analysis

The screenshot shows a statechart editor interface. The main canvas displays a statechart with states Start, A, B, and End. Transitions are labeled with actions and guards: 'start [] counter := 0;', 'go-to-B [] counter := counter + 1;', 'back-to-A [counter < 11]', and 'end [counter > 10]'. A variable 'Integer:counter' is shown. Red arrows point to specific elements with labels: 'compute transition ends from labels' points to the transition from A to B; 'compute closure' points to the state A; 'reuse of metamodels and semantics' points to the statechart diagram and the palette. The palette on the right lists metamodel elements: do. Action, FinalState, InitialState, NormalState, and StateMachineVariable. The bottom panel shows the 'NormalState' properties table.

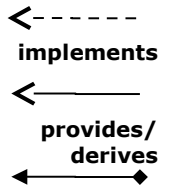
Core	Property	Value
Appearance	Id	0
	Label	A
	Reachable	Normal State 1, Normal State 0, Final State 5
	Successors	Normal State 1



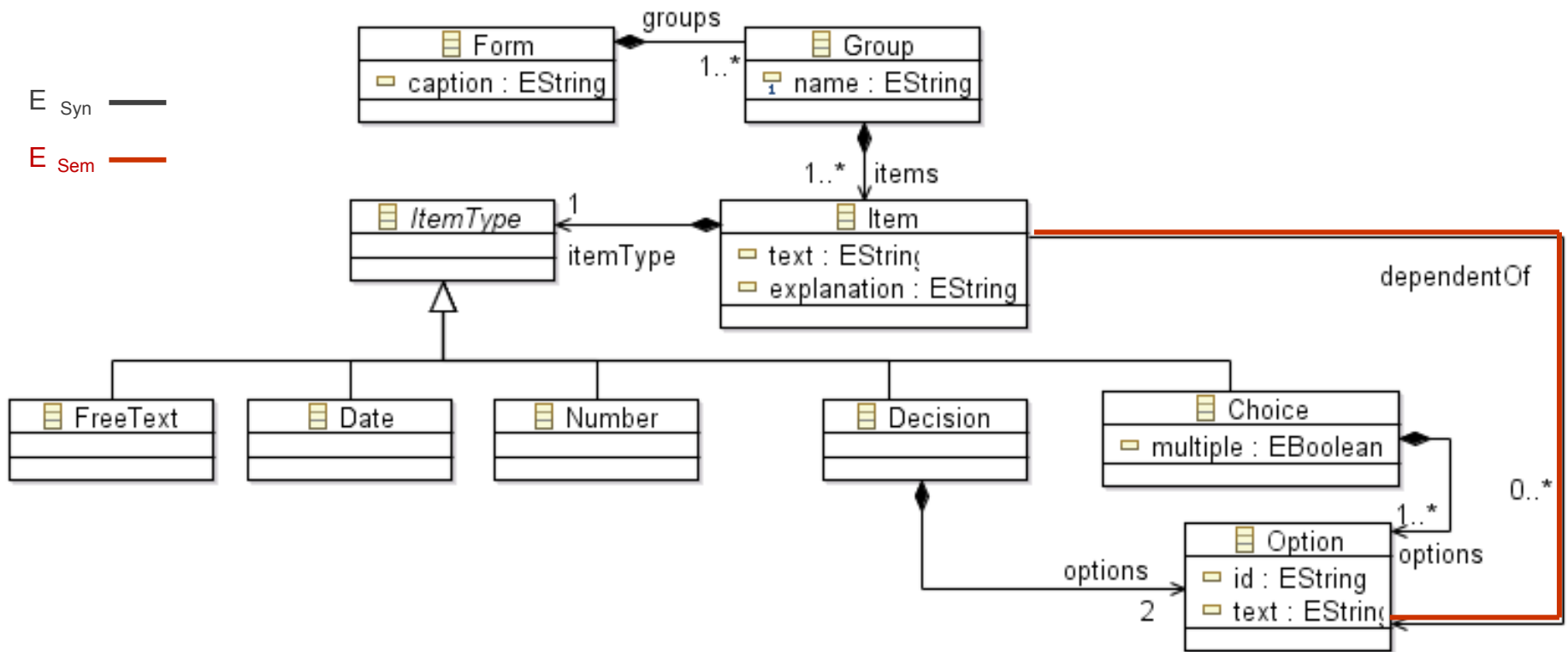
Example 2: Forms DSL



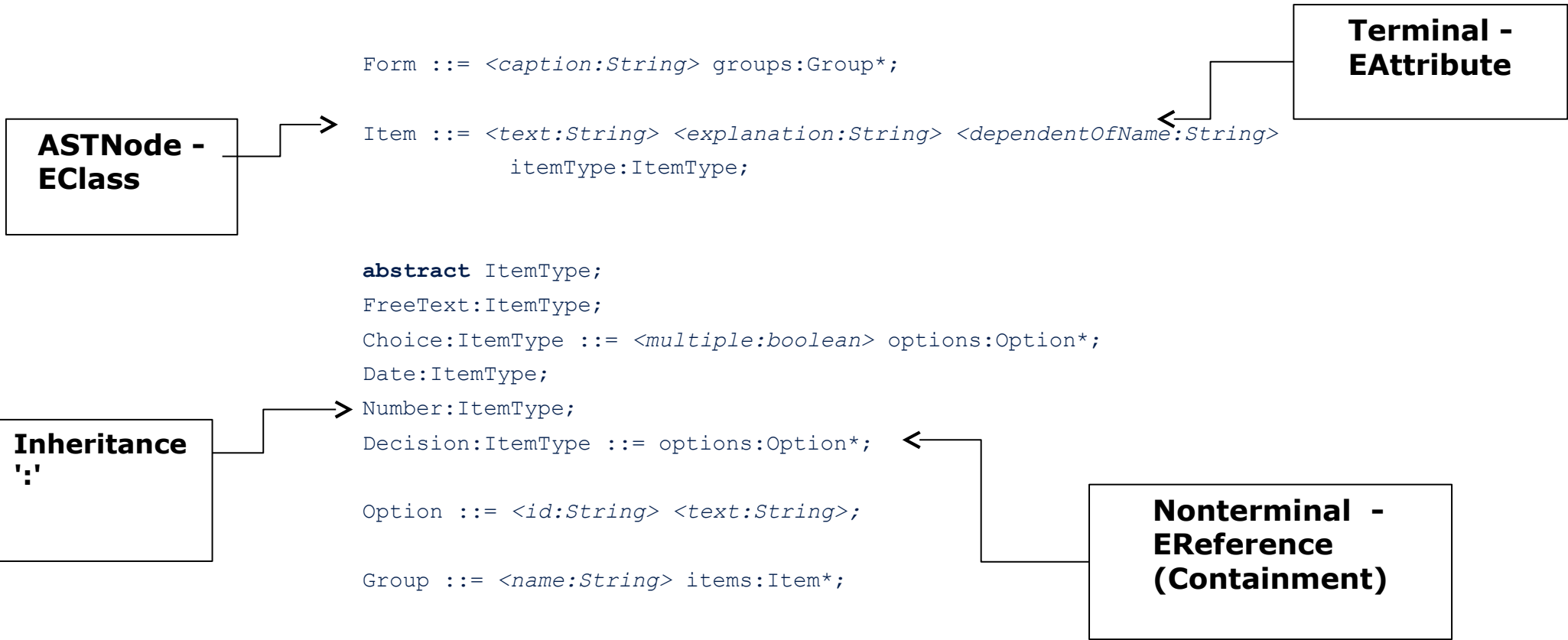
references/uses



Example 2: Forms Metamodel



Example 2: Forms as AST Grammar



Example 2: Forms Attributes

```
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)

The screenshot displays the Eclipse Forms Editor interface. The main editor window shows the following form definition:

```
FORM "GTTSE'11 Questionnaire"  
  GROUP "General Questions"  
    ITEM "Name" : FREETEXT  
    ITEM "Age" : NUMBER  
    ITEM "Gender" : CHOICE ("Male", "Female")  
  
  GROUP "Research Program"  
    ITEM "Do you enjoy the GTTSE'11 research program?"  
      : DECISION ("Yes", "No")  
  
    ITEM "How many tutorial have you attended so far?"  
      : NUMBER  
  
  GROUP "Food and Drinks"  
    ITEM "Food Preferences"  
      : CHOICE ("All", "Vegetarian", "Vegan")  
  
    ITEM "Only non-achoholic drinks?"  
      : DECISION ( no_alcohol:"Yes", alcohol:"No")  
  
    ITEM "Does the menu match your eating preferences?"  
      : DECISION ("Yes", "No")  
  
    ITEM "Do you like Vinho Verde?"  
      ONLY IF alcohol  
      : CHOICE ("It's great!",  
              "It's great for lunch!",  
              "It's OK.")
```

The Outline view on the right shows the hierarchical structure of the form:

- Form GTTSE'11 Questionnaire
 - Group General Questions
 - Item
 - Item
 - Item
 - Group Research Program
 - Item
 - Item
 - Group Food and Drinks
 - Item
 - Item
 - Item
 - Item alcohol
 - Choice
 - Option
 - Option
 - Option

The Properties view at the bottom shows the following details for the selected item:

Property	Value
Dependent Of	Option alcohol
Dependent Of Name	alcohol
Explanation	
Text	Do you like Vinho Verde?

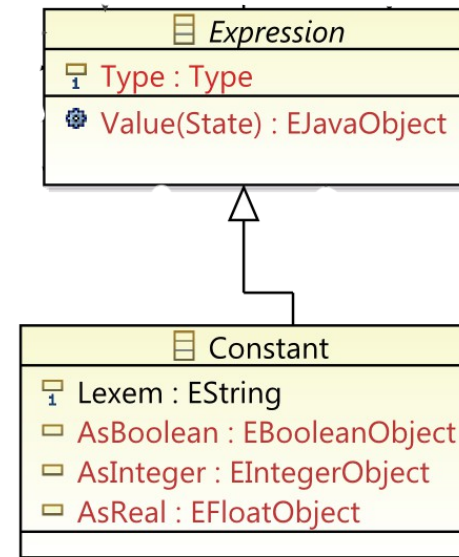
Example 3: SiPLE Types (Excerpt from Semantic Interface)

```
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();
}
```

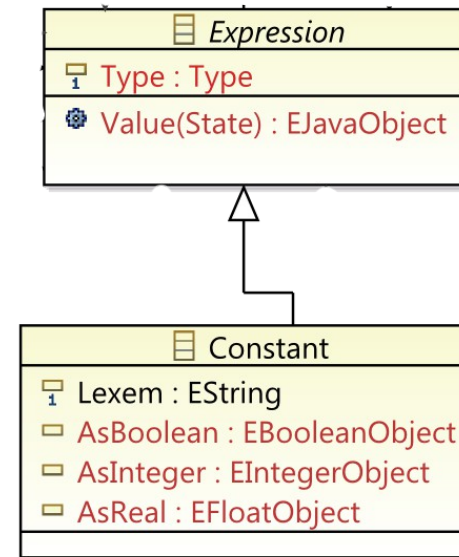
Example 3: SiPLE Types (Excerpt from Definitions)

```
/** 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;  
}
```



Example 3: 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 Editor

The screenshot displays the Eclipse IDE interface for the SiPLE Editor. The main editor window shows the source code for `test.siple` with the following content:

```
Procedure main() Begin
  x := 0;
  y := 0;
  z := 0;

  Var x:Integer;
  x := 100;

  Procedure writeMain() Begin
    Write x;
    Write y;
    Write z;
  End;

  Procedure l1() Begin
    Var y:Integer;
    y := x + 100;

    Procedure writeL1() Begin
      Write x;
      Write y;
      Write z;
    End;

    Procedure l2() Begin

```

The Outline view on the right shows a hierarchical tree structure of the code:

- Compilation Unit
 - x
 - y
 - z
 - writeGlob
 - Block
 - incGlobVars
 - Block
 - Variable Assignment Integer
 - Variable Assignment Integer
 - Variable Assignment Integer
 - main
 - Block
 - Variable Assignment Integer
 - Variable Assignment Integer
 - Variable Assignment Integer
 - x
 - Variable Assignment Integer
 - writeMain
 - l1
 - Procedure Call Undefined

The Properties view at the bottom shows the following table:

Property	Value
As Boolean	
As Integer	100
As Real	
Lexem	100
Procedure In Context	Procedure Declaration main
Type	Integer

The status bar at the bottom indicates the current state: Writable, Insert, and line 28:9.

RAGs are only well-suited for analysis of modes, if the metamodel does not specify a degenerated tree structure.

Degenerated 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

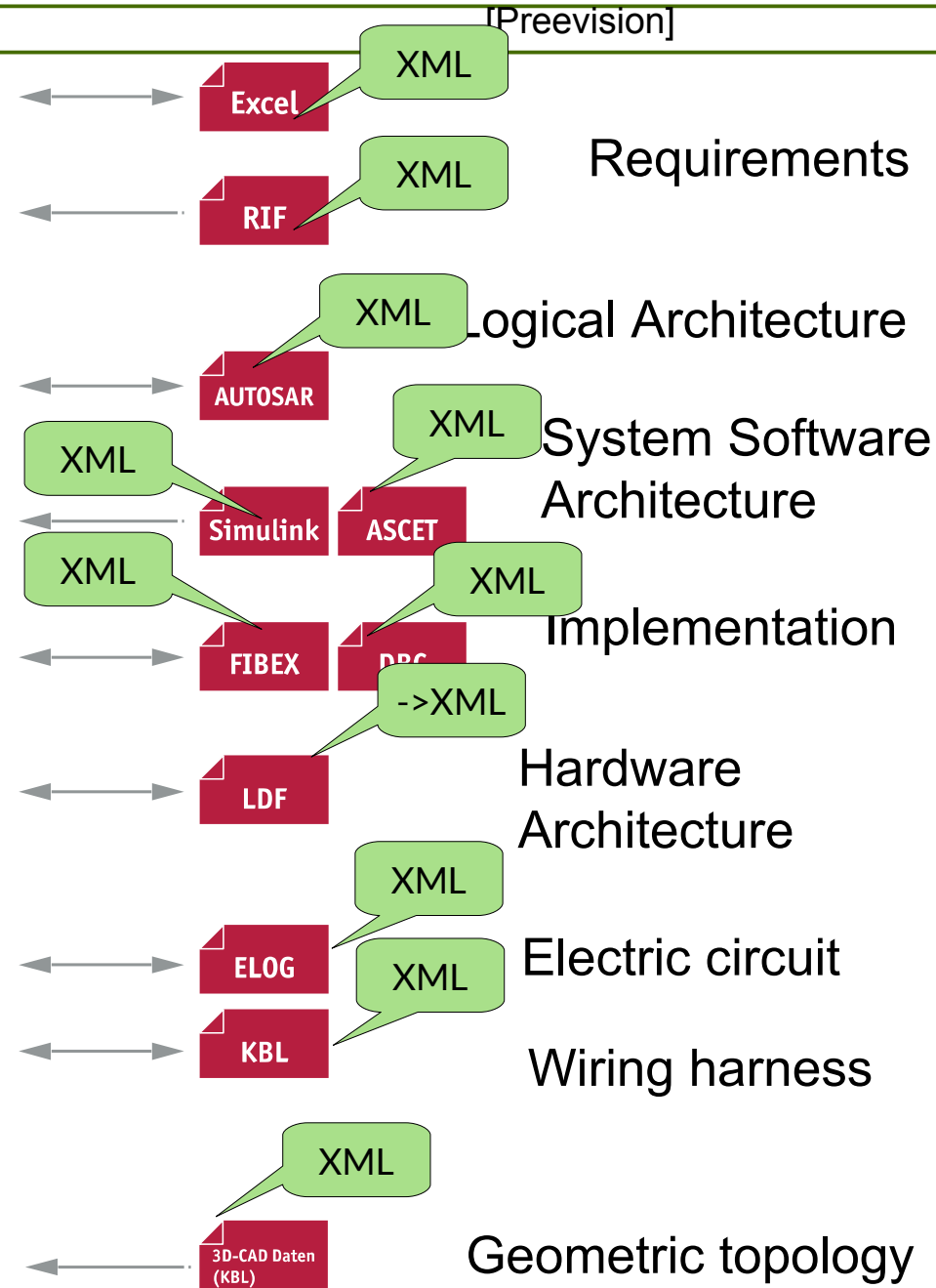
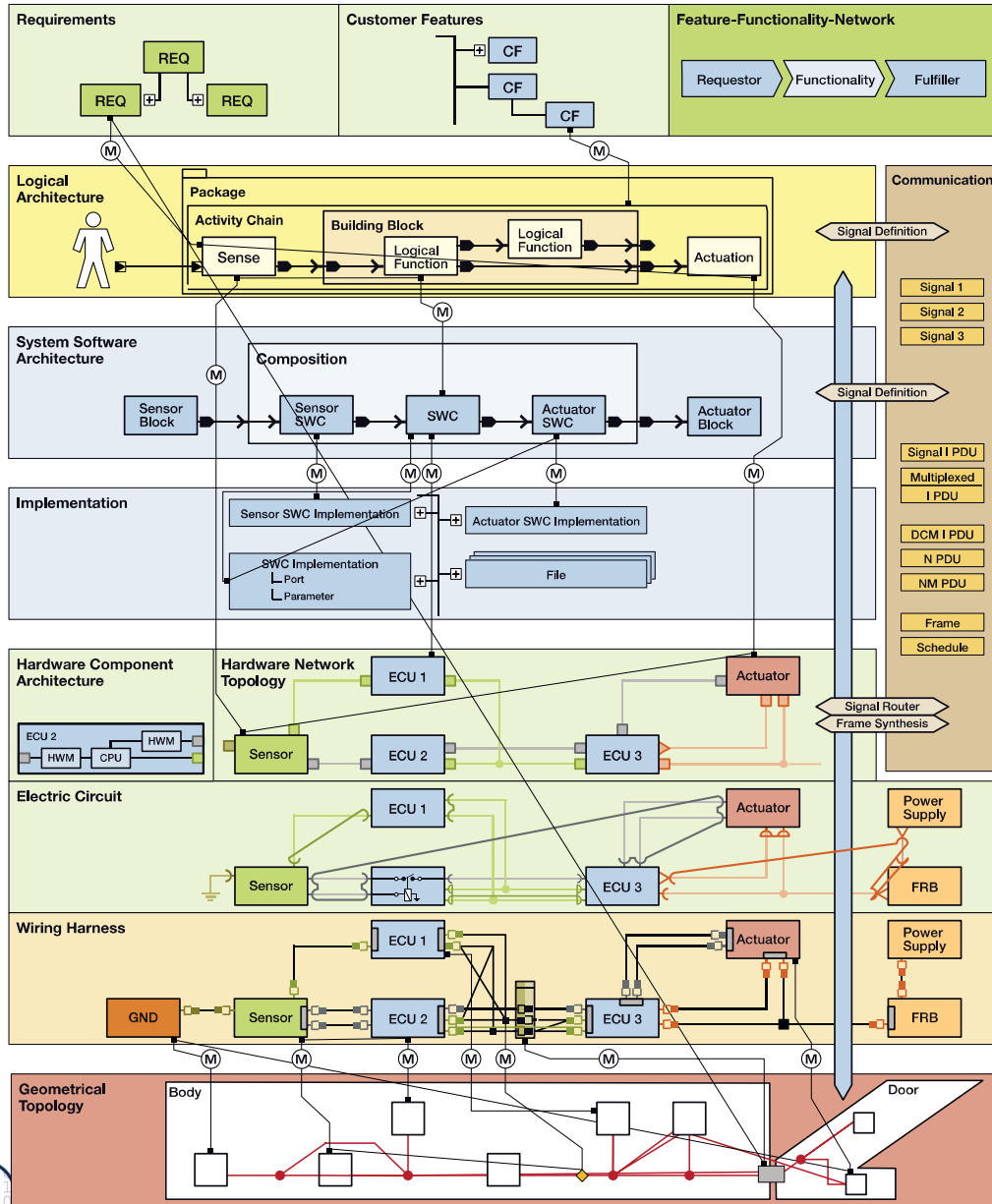
Conclusion

- ▶ Common metamodels specify **link-tree structures** enriched with semantic interfaces (e.g. EMOF, MOF).
- ▶ **RAGs can be used to specify 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 manuel changed attribute values)
 - Incorporation of JastAdd's rewrite capabilities
 - Working on reducible graphs
 - JastEMF is still under development by Sven Karol (Jeronimo Castrillon's Chair of Compiler Construction)

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

- ▶ Of an MDSD application

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



Links on the XML Formats of PreeVision

- ▶ Excel:
<https://support.office.com/de-de/article/%C3%9Cberblick-%C3%BCber-XML-in-Excel-f1>
- ▶ RIF: https://en.wikipedia.org/wiki/Requirements_Interchange_Format
- ▶ Simulink:
<http://de.mathworks.com/help/rptgenext/ug/how-to-compare-xml-files-exported-from-s>
- ▶ 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>
 - <http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templ>
 - 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>
 - http://www.etas.com/download-center-files/products_ASCET_Software_Products

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

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