Applying Attribute Grammars for Metamodel Semantics

Christoff Bürger[2]
in cooperation with Sven Karol and Christian Wende
{Christoff.Buerger@tu-dresden.de}

ECOOP, FML Workshop, Maribor, 21.06.2010

[2]Gefördert durch ein Stipendium des ESF und des Freistaates Sachsen

TECHNISCHE UNIVERSITÄT DRESDEN

Contents

01 Motivation

Metamodelling, Compiler Construction and AGs

Metamodelling is a standardisation process to support

- Tool and repository generation, integration and reuse
- A standard Framework with common functionalities and support tools (Generators; Editors; Persistency; Transformation and query languages)

⇒ Metamodels specify a common tool/repository API

Compiler construction is about the well-founded specification and development of reliable, efficient, complete language processors.

- AGs: **Specification** of static semantics
- AG Systems: Generation of evaluator **implementations**

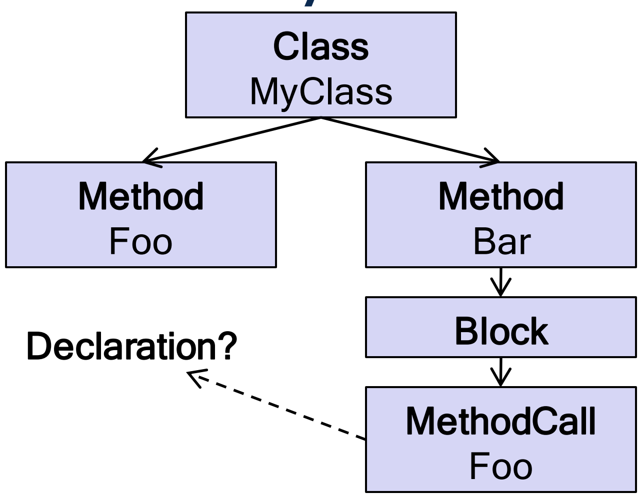
Metamodelling and compiler construction complement each other.

01 Motivation
The Need for Metamodel Semantics

Problem: Metamodels‵ semantic gap

- Specify a language‵s structure enriched with semantic interfaces
- Do not specify the language‵s semantics

Example: Name analysis



⇒ Many possible name resolutions
⇒ Can be complicated (Shadowing, overloading, several namespaces, namespace modifiers e.g. super, etc.)

01 Motivation
The Need for Metamodel Semantics

Question: What is the meaning of a metamodel?

⇒ Semantics is implemented manually in generated code
⇒ Semantics is specified in operational languages (e.g., Kermeta)
⇒ Semantics is encoded in metalmodel based tools
- Graphical editors, model transformers, composition programs, visitor based interpreters

⇒ However: Semantics should be specified by appropriate formalism

Idea: Use existing well-known formalisms from compiler theory and practice (AGs) to specify semantics.

01 Motivation
A few general Words about Semantics

Semantics is

- Always specified w.r.t. well defined structures
- To reason about structures to derive information or extend/manipulate it

The complicated part of semantics is

- The distribution of local information w.r.t. constraints accross the structure
- To combine such information and
- Further redistribute the results

$\Rightarrow$ In practice, reasoning w.r.t. a local context is trivial

AGs are very convenient to specify semantics for <u>tree structures</u>, if the structure is not changed or only extended.

02 The JastEMF Approach

Metamodelling Languages, Tree Structures and AGs

Claim:

Most metamodeling languages�much 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* 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?

02 The JastEMF Approach
Ecore: Tree Structure and Semantics

Each model instance of an Ecore metamodel has a spanning tree

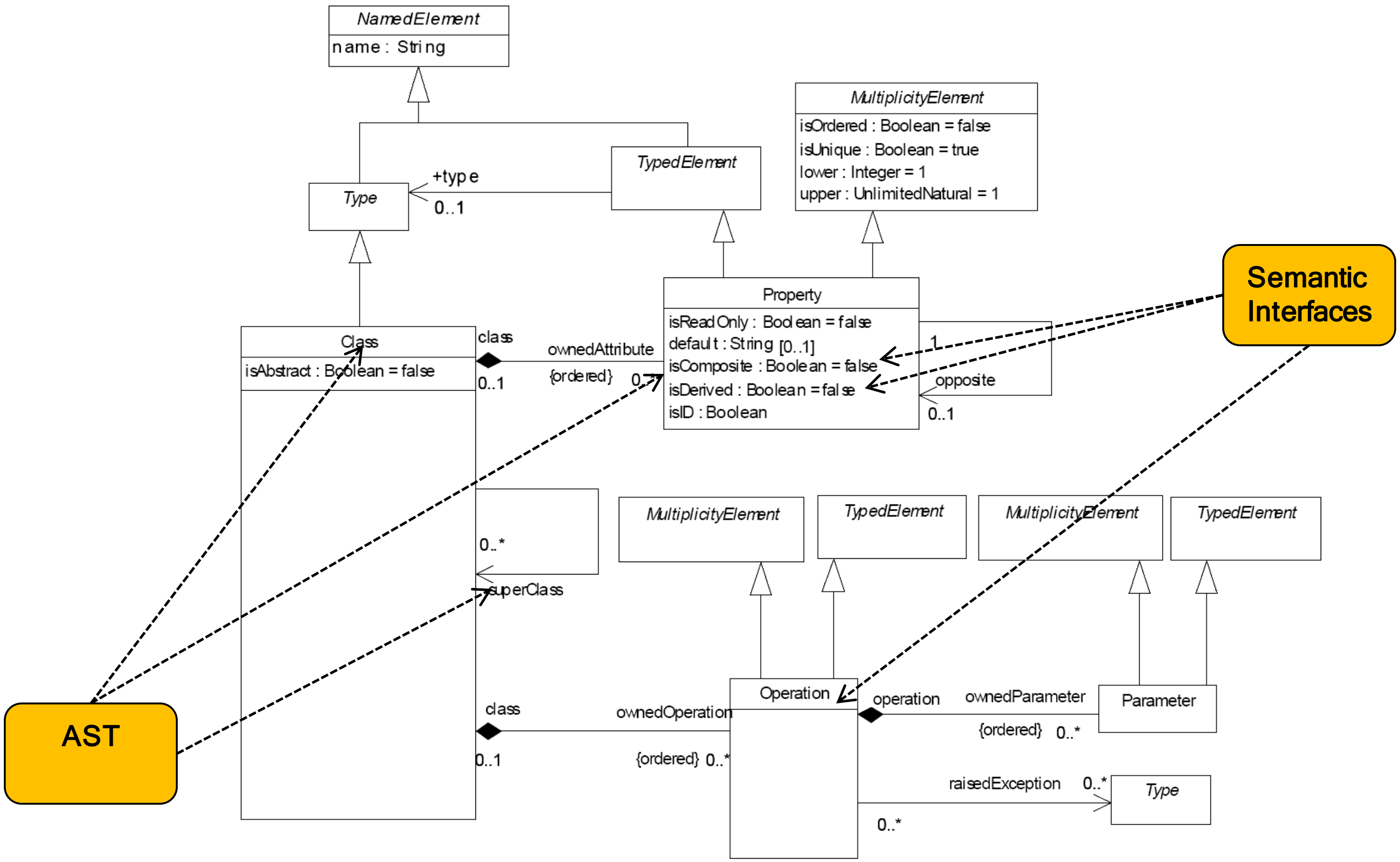
- Its set of nodes are all metaclass instaces (Non-terminals) and non-derived properties (Terminals)
- Its edges are metaclass instances‛ containment references

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.

02 The JastEMF Approach



02 The JastEMF Approach

JastAdd: AST and Attribute Specifications

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.

02 The JastEMF 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:

```
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;
```

02 The JastEMF Approach
Ecore-JastAdd Concept Mapping

In summary: EMF and JastAdd generate a class hierarchy

- EMF
  - Metamodel implementation (Repository + Framework/Editors etc.)
  - 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

02 The JastEMF Approach
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

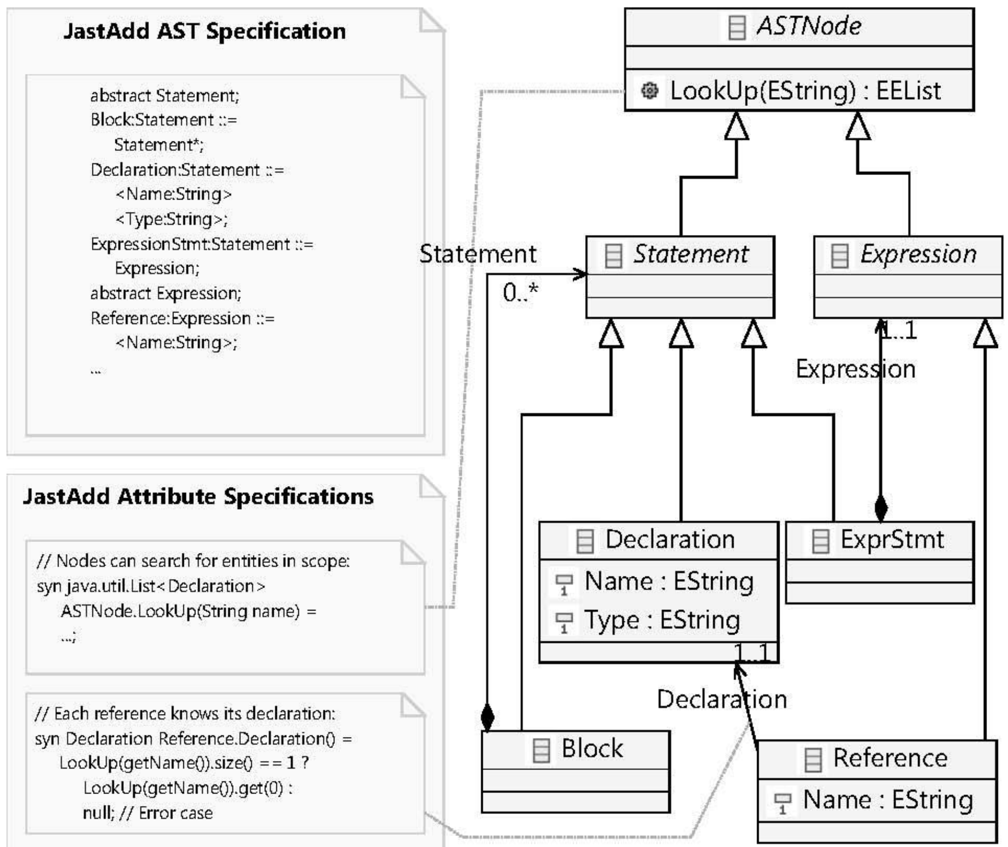
02 The JastEMF Approach

Ecore-JastAdd Concept Mapping

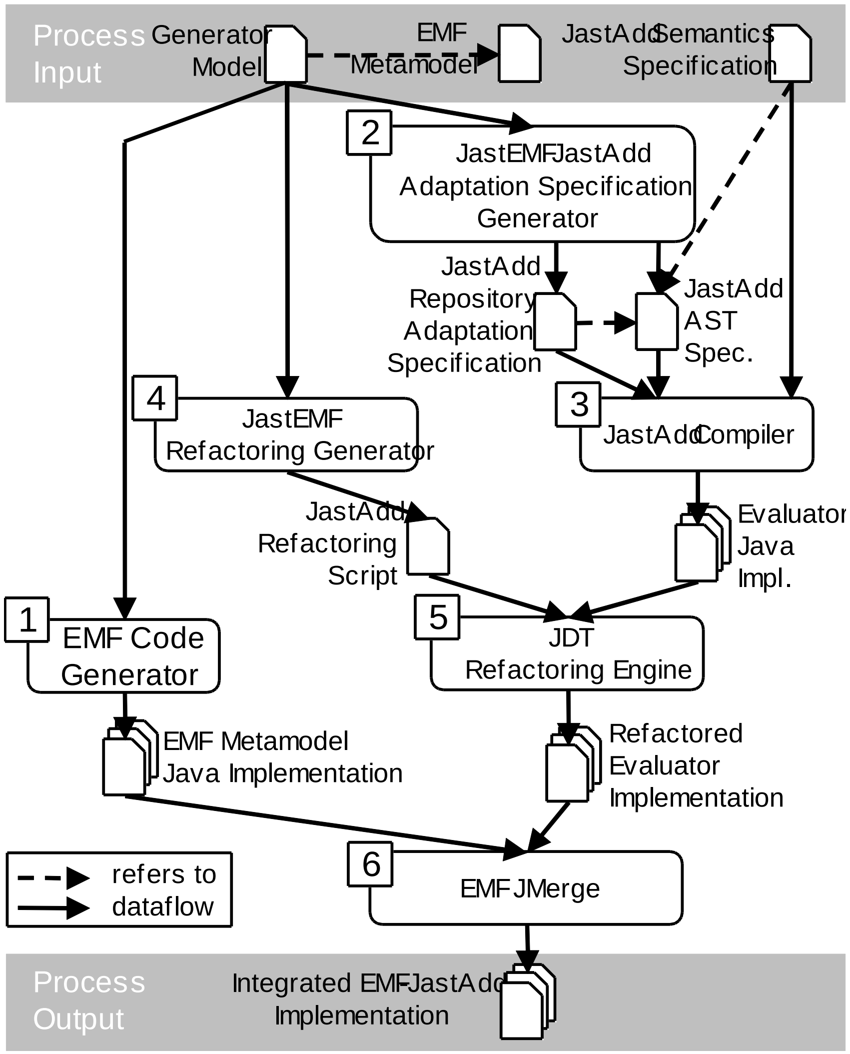
AST node types	EClasses
AST terminal children	EClass non-derived properties
AST non-terminal children	EClass containment references
Synthesized attributes	EClass derived properties
	EClass operations
Inherited attributes	EClass derived properties
	EClass operations
Collection attributes	EClass properties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

02 The JastEMF Approach

Ecore-JastAdd Concept Mapping



JastEMF's Integration Process



⇒ JastEMF steers EMF & JastAdd

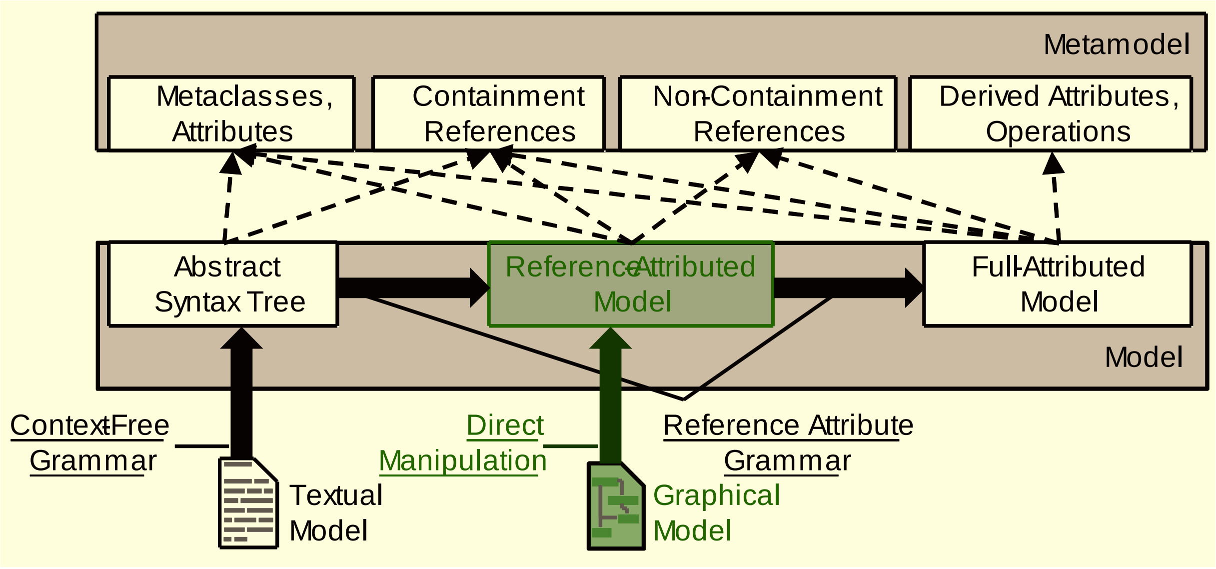
⇒ EMF and JastAdd development can be handeled as used to

TECHNISCHE UNIVERSITÄT DRESDEN

02 The JastEMF Approach
A few Words about Graphs

Semantic evaluation can start from (partly) reference-attributed 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



03 Remarks, Conclusion and Outlook
Important Remark

RAGs are only <u>well-suited</u>, 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

03 Remarks, Conclusion and Outlook
Conclusion

Common metamodelling languages‘ metamodels specify tree structures enriched with semantic interfaces (e.g. (E)MOF).

RAGs can be used to specify static semantics for such metamodels.

JastEMF *(www.jastemf.org)*: **Tool to generate semantic metamodel implementations based on Ecore metamodels and JastAdd AGs.**

03 Remarks, Conclusion and Outlook
Outlook

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

Further examples beside SiPLE: Statemachine DSL (Already Implemented)

- Graph language example (EuGENia generated GMF editor)
- Better example w.r.t. the metamodelling world
- Reuses SiPLE for transition actions, guards and entry actions
- Has SiPLE code generation

$\Rightarrow$ Executeable