TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

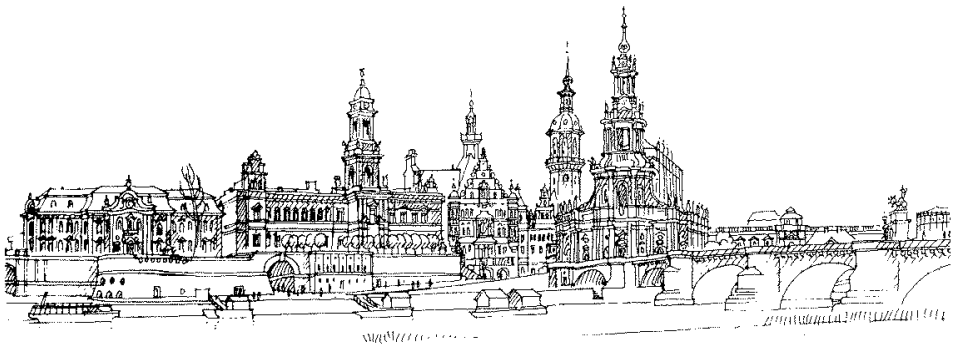
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI10-03-März 2010

Christoff Bürger, Sven Karol

Institut für Software- und Multimediatechnik

Towards Attribute Grammars for Metamodel Semantics



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/

Towards Attribute Grammars for Metamodel Semantics

Christoff Bürger and Sven Karol

Institut für Software- und Multimediatechnik
Technische Universität Dresden
D-01062, Dresden, Germany
{Christoff.Buerger, Sven.Karol}@tu-dresden.de

Abstract. Of key importance for metamodelling are appropriate modelling formalisms. Most metamodelling languages permit the development of metamodels that specify tree-structured models enriched with semantics like constraints, references and operations, which extend the models to graphs. However, often the semantics of these semantic constructs is not part of the metamodel, i.e., it is unspecified. Therefore, we propose to reuse well-known compiler construction techniques to specify metamodel semantics. To be more precise, we present the application of reference attribute grammars (RAGs) for metamodel semantics and analyse commonalities and differences. Our focus is to pave the way for such a combination, by exemplifying why and how the metamodelling and attribute grammar (AG) world can be combined and by investigating a concrete example — the combination of the Eclipse Modelling Framework (EMF) and JastAdd, an AG evaluator generator.

1 Motivation

Modelling has always been a key activity in software engineering [26]. Abstracting from the real world (i.e., extracting a representation of a domain's elements and their relationships) to become able to reason about it is a basis for successful software development. A common understanding — i.e. metamodel — is a necessity to be able to reason about information provided by a third party, to manipulate it and to support it for others. That is why metamodels are important. They specify a common interpretation of their model instances and permit the integration of tools based on a common repository. To achieve such an unique understanding of models, a metamodel's specification must be complete with respect to the question if some input is a model and, if it is, how the model looks like[1]. Otherwise, tools sharing the same metamodel can hardly cooperate, since each tool may interpret model instances differently.

[1] It is important to acknowledge, that only the specifications of metamodels must be complete. The actual algorithms implementing a metamodel's specification, i.e., how to recognize model instances and how models are (physically) represented, do not have to be part of it.

If we consider metamodels used in practice, a problem becomes obvious. As will be shown in Section 2.1, most metamodelling languages support the specification of an abstract syntax underlying each model instance — a spanning tree — which is enriched with semantic constructs like constraints, references and operations, that impose a graph on top of it. Unfortunately, it is impossible in most metamodeling languages to develop metamodels that not only specify each model's spanning tree and declare the existence of a graph, but also specify the graph's shape. Thus, semantic constructs like constraints, references and operations are often not defined, but rather only declared. Consequently, even if tools based on such a metamodel agree on how to interpret a model's context-free structure, the interpretation of its embedded semantics may disagree. Such metamodels only specify syntax enriched with semantic interfaces and lack formal semantics. For example, consider a declared reference of a certain type. The metamodel developer might have specific visibility restrictions for the reference in mind, but he cannot specify them within the metamodel. Consequently, each of the cooperating tools has to implement its own well-formedness checks or name resolution algorithms. As a more concrete example, consider the EMF's Unified Modelling Language (UML) metamodel [8] which specifies the existence of types, but does not specify (i.e., include) the resolution of type references. Assume a refactoring tool is used to change the type of an entity. Since another tool based on the same metamodel may perform a different type resolution, the refactoring tool's changes may obtain a completely different meaning. Similar problems occur for operations which are declared, but whose functionality cannot be specified within the metamodel.

How to best represent semantics in metamodels and to what extent this is feasible is still an open issue. For now, it is important to acknowledge that at some point semantics must be specified regardless of whether semantics is part of metamodels or is tool dependent. And it would be very pleasant, if metamodel semantics can be specified using a formalism.

In the remainder of this paper we investigate the application of the well known AG formalism [17,18], extended by reference attributes [14], for the specification of metamodel semantics with respect to practical issues. The next section shortly introduces metamodels and AGs. Both are related to each other to clarify why AGs are appropriate to specify metamodel semantics. Afterwards, a concrete prototype integration is presented: The adaptation of JastAdd AG evaluators to the EMF. Encountered problems are explained and general issues of the concept are highlighted. A section about related work follows. Finally, a summary and outlook conclude the paper.

2 Metamodels and Attribute Grammars

AGs are a well-known formalism to specify semantics for context-free languages. If we like to specify metamodel semantics using AGs, metamodels must implicitly specify context-free structures, i.e. a spanning tree for each model instance. We think, that this is the case and, that consequently RAGs are an appropriate

formalism to specify metamodel semantics. In the rest of this section these claims are substantiated. Since no "consolidated body of knowledge" about the design of models, metamodeling languages and the application of model driven techniques exists yet [26], we investigate a representative metamodelling language — the Essential MOF (EMOF) [11] — to exemplify, that model instances satisfying a metamodel M developed in a common metamodelling language indeed have a spanning tree implicitly specified by M. Based on that observation we introduce AGs and their application for metamodel semantics.

2.1 Metamodelling Languages

In this section, we first discuss general concepts of metamodelling languages. Afterwards, a distinction between metamodels' syntactic and semantic constructs is made based on the general concepts' characteristics. Finally, we show thir application in EMOF.

General Metamodelling Language Concepts

In the domain of metamodelling languages certain concepts — thus, meta-metamodel constructs and their relationships — are common. Most metamodelling languages support a tailorable, nameable basic construct used to model arbitrary domain concepts[2]. The tailoring of such a basic construct C_M, to represent a certain *domain concept* C_D and its *intrinsic* and *extrinsic* values[3], is often achieved using nameable *property* constructs associated with C_M, whereas each value of C_D is represented by one property. To specify a property's co-domain, metamodelling languages support the concept of types. Each type represents a certain co-domain and most metamodelling languages have a common set of basic types like integers, floating point numbers, strings and enumerations. Domain concepts, i.e., basic constructs and their properties, are also types, since they abstract from (possibly infinite) sets of distinct entities. To permit the computation of several extrinsic values in one step, the explicit computation of context information and model manipulations, many metamodelling languages support a typed *operation* construct. Using typed properties, *relationships* between domain concepts can be modeled. To model a directed relationship between a domain concept C_1 and C_2, a property of type C_2 associated with C_1 can be used. Alternatively, some metamodelling languages have an *association* construct to model relationships more explicitly. Anyway, most metamodelling languages support one special relationship between domain concepts — the *composition*. If entities of a domain concept C_1 consist of entities of a domain concept C_2, such that the C_2 entities become an inextricable part of the C_1 entities, the relationship between C_1 and C_2 is a composition.

[2] domain concept = concept of the domain for which a metamodel is developed

[3] An extrinsic value depends on the concrete relationships and values of properties its associated entity has — it depends on its context.

Model Syntax and Semantic

An important observation is that iff an entity E_1 is a composite of an entity E_2, E_2 cannot be a composite of E_1. Thus, the graph of any valid model instance's composition relationships must be a tree[4]. This does not imply that metamodels' composition relationships must be acyclic. The correlation between the tree shape of model instances and the graph shape of their metamodels is analoug to the one between a concrete word derivable from a context-free grammar (CFG) and the grammar itself. The word's structure is a tree, even if the dependency relationship between the grammar's non-terminals can be an arbitrary graph. In addition, the *generalisation* concept supported by most metamodelling languages does not contradict this observation — a metamodel's composite relationships specify a spanning tree for its model instances[5]. Contrary, metamodels' *non-composite relationships* only declare the existence of relationships between domain concepts, but do not specify them. This means, although models have a spanning tree containing all entities, a non-composite relationship R between two domain concepts C_1 and C_2 still does not specify with which concrete entity $E_2 \in C_2$ R associates an entity $E_1 \in C_1$. Non-composite relationships can depend on any known model instance information, in the simplest case just intrinsic property values like ids up to complicated context dependencies — computable based on the model's spanning tree — or a combination thereof. With respect to the section's introduction — that the reasoning about a model instance's structure is its semantics — non-composite relationships represent model semantics. Since metamodels do not define non-composite relationships' semantics, they are only *semantic interfaces*. In addition, extrinsic values and operations are semantic interfaces, since their context dependencies and manipulations are not specified within metamodels too.

In consequence, we say that most metamodelling languages only permit the declaration of extrinsic values, non-composite relationships and operations, but do not support their definition. They have no method to specify such constructs' value and behaviour — they do not support their specification. Therefore, we regard metamodels' basic constructs (representing domain concepts), composite relationships and intrinsic values as syntax and their extrinsic values, non-composite relationships and operations as semantic.

---

[4] To be precise, the graph of any valid model instance's composition relationships must be acyclic. It is still possible, that an entity E_C is a composite of two distinct other entities E_1, E_2. E.g., a door is part of two rooms. However, in such cases the entity can be considered to be the composite of just one of the entities E_1 or E_2 and have an ordinary relationship, resolved by its semantics, with the other one. It is only important, that the overall structure, the model's semantics are based on, is well-defined.

[5] Since most metamodelling languages L are specified using a metamodel $M \in L$ — a meta-metamodel — like CFGs are specified using a CFG, also metamodels have a spanning tree. Thus, our presented approach is feasible to specify the semantics of metamodelling languages.

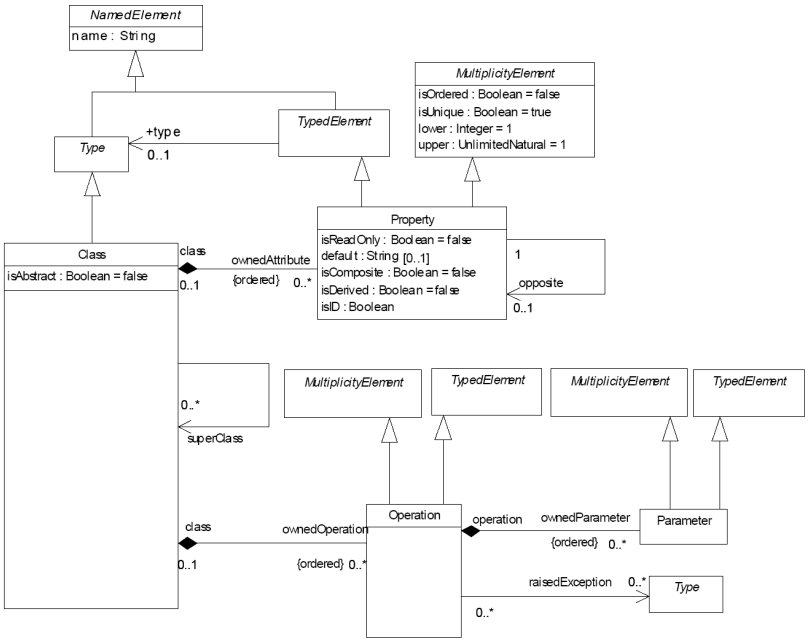


Fig. 1. EMOF core constructs [11]

EMOF as Example Metamodelling Language

Figure 1 illustrates the concepts introduced above using EMOF's specification — its meta-metamodel. The basic construct used to model domain concepts is called `Class` and properties simply `Property`. The relationship between `Class` and `Property` is a composition and the properties of a certain domain concept are its owned attributes. In a similar way, operations are modeled. Properties can model intrinsic and extrinsic values, depending on their `isDerived` value. They are also used to model relationships — called associations — in which case they can be a composite depending on their `isComposite` value. Finally yet importantly, a special `superClass` association of `Class` is used to model generalization.

2.2 Attribute Grammars

AGs are a form of two level grammar [17,4,19], i.e., they specify semantics on top of context-free structures. Usually, their basic context-free structures are abstract syntax trees (ASTs), which are specified using some kind of CFG. To reason about CFGs synthesized and inherited attributes are associated with non-terminal symbols. A synthesized attribute represents an information flow upwards the syntax tree. Thus, a synthesized attribute's equation — i.e., the specification how to compute it — can depend on any information given in

the (sub-) tree the node the attribute is associated with spans. Analogously, an inherited attribute's equation represents an information flow downwards the syntax tree. Using synthesized and inherited attributes context information can be freely distributed and combined across the AST. Hence, AGs provide an elegant formalism to specify semantics of context-free languages.

However, the basic formalism as introduced by Knuth [17,18] does not provide sufficient concepts to be applicable for object-oriented structures. Therefore, several AG extensions were developed. In [13], attribute grammars are applied to object-oriented ASTs supporting generalization, which is also a key concept of metamodelling languages. Additionally, reference attributes (RAGs, [14]) allow direct information flows between remotely located AST nodes and can be used to easily specify an arbitrarily structured abstract syntax graph (ASG) on top of the AST. Finally, circular attributes [7] ease the specification of complicated semantic computations.

Besides being declarative, AGs have a very good tool support. Broad ranges of compiler construction environments use RAGs to implement their semantics. Furthermore, many AG tools permit the separation of semantic concerns (e.g., type analysis or data flow analysis) as well as semantic extensibility [5]. A very convenient characteristic of AGs is that specifications can be checked for consistency and completeness, i.e., static checks if semantics is well defined for any AST. Finally yet importantly, AGs are well understood, investigated for a long time and still have an active research community.

2.3 Unifying Metamodels and Attribute Grammars

Using RAGs for Metamodel Semantics

Considering the last two sections about metamodelling languages and RAGs, a concept how to apply RAGs to specify metamodel semantics becomes obvious. If a metamodel M is modelled as described in Section 2.1, M separates all its model instances M_i into two parts: A context-free tree structure S, specified by M's composite relationships, and a declared, but not defined, graph structure imposed on top of S, given by M's derived properties, non-composite relationships and operations. Each M_i's imposed graph structure represents its semantics. If one likes to specify semantics for all possible model instance M_i using an object-oriented RAG, M's composite relationships can be used to derive an appropriate AST specification for the RAG — i.e. a specification that accepts any M_i's S. Additionally, each of M's derived properties, non-composite relationships and operations must be defined by the RAG. This can be achieved by representing each derived property by an attribute of equal type, each non-composite relationship by a reference attribute of equal destination type and each operation by a parameterised attribute with equal signature, whereas each of these attributes is associated with the AST node type representing the concept they are associated with in M. The desired metamodel semantics (i.e. each model instance's semantic) is now the ASG the RAG implies on top of each M_i's S.

Using RAGs for Graph-Shaped Models' Semantic

By applying RAGs to metamodels, we argue that one can go beyond this strict separation between the context-free and imposed graph structure by also considering graphs, with unique spanning trees, as possible starting points for attribute evaluations. The only condition beside the spanning tree is, that nodes of the same type always have the same set of reference types. Edges in such an input graph, that are not part of its spanning tree, can be considered as reference attributes with a predefined value and destination type. As presented before, EMOF metamodels specify unique spanning trees for their model instances and non-containment references are typed and associated with a `Class`. Thus, any EMOF metamodel satisfies the mentioned conditions. In fact, to have graphs as input for semantic evaluations is rather common, since the EMF provides a set of standard reference resolution algorithms, e.g. to resolve the reference values of models serialized as XML files.

A Methodology for Metamodel Development Waiting in the Wings

The presented approach is not only theoretically backed and technically and practically convenient but also provides a kind of methodology how to develop metamodels with semantics. Its key concept is to separate a metamodel into its context-free parts and its semantics. In a first step, it has to be clarified how entities' syntax looks like. In a second step, the semantics of simple relationships has to be developed, such that models' basic reference graph is well-defined. References permit the distribution of information known to one model entity to arbitrary other entities and consequently ease the specification of complicated model semantics. Finally, the desired metamodel semantics can be specified, reusing the imposed reference graph.

3 Example Metamodel and Attribute Grammar Integration

In this section, we discuss the integration of metamodelling and attribute grammar technology by the example of the EMF and the JastAdd meta-compiler. The first two subsections briefly introduce the features of both tools that are important for their integration. Afterwards, the integration approach is described and a mapping between EMF features and JastAdd features is presented. Furthermore, we analyse arising conflicts, which we distinguish in groups of structural or behavioural.

3.1 The Eclipse Modelling Framework

The EMF is a common modelling framework for Java. Its core language Ecore roughly corresponds to EMOF and supports the development of metamodels

by graphical editing in a UML class diagram like notation. Essentially, Ecore-based metamodels define the abstract syntax of a language. Like EMOF, the language does not provide first class language constructs to specify semantics. Consequently, the most common way to implement the semantics of Ecore models is adding Java code to the generated metamodel classes or encode it in the applications relying on the metamodel. To address this problem, several tools, mostly not based on a well-known semantic formalism like AGs, have emerged. We compare some of these tools to our approach in Section 4.

In comparison to EMOF, Ecore has some advantages that ease its extension with AG features and its integration with JastAdd. First, it is not only a standard, at least for Java the EMF provides intuitive code generation facilities. Metamodel classes (`EClass`) are generated as Java interfaces and Java implementation classes. Their properties become class attributes that can be accessed via getter and setter methods and packages (`EPackage`) become Java packages. A further important advantage over EMOF is a more sophisticated distinction between attributes (`EAttribute`) and references (`EReference`). References can additionally be classified as *containments,* defining context-free structure, or *non-containments,* defining graph structure. To reduce confusion of ideas we refer to Ecore attributes as properties for the rest of this section.

3.2 The JastAdd Metacompiler

JastAdd [1,6] is an object-oriented, AG-based system to specify language tooling reaching from type and flow analyses to whole compiler frontends. It allows generating demand-driven Java AG evaluators. Besides the basic attribute grammar concepts, JastAdd supports advanced concepts such as reference and circular attributes.

A JastAdd specification consists of several modules. The abstract syntax of the target language is specified by an AST specification that consists of a list of node types (*non-terminals*) and a list of child nodes (*terminals* or *non-terminals*) for each of them. In the following a small example AST specification is given wich defines the abstract syntax of simple addition expressions. It contains an abstract node type `Expression` and two concrete realisations. `Addition` has two expression children and expectedly represents binary arithmetic additions. A `Constant` declares two terminals defining its type (which may be `Real` or `Int`) and value (which may be a real or integer value).

```
abstract Expression;
Addition:Expression ::= Op1:Expression Op2:Expression;
Constant:Expression ::= <ConstValue:Value> <ConstType:Type>;
```

Semantics is usually specified within several modules containing attribute definitions and attribute equations, which are associated with node types in the AST specification. In the following, we specify a type analysis and evaluation methods for the addition language above. Two synthesised attributes *Type* and *Value* for the `Expression` non-terminal are declared using the keyword *syn*. Since `Expression` is abstract, we only specify the corresponding equations for

both of the concrete node types using the keyword *eq*. For `Constants`, we just need to pass the corresponding terminal values to the attributes. The type of `Additions` depends on their operands' types. If both of them are of type `Int` the `Addition's` type is also defined as `Int`, otherwise it is defined as `Real`. For value computation, an appropriate *sum* implementation is selected depending on the value of the *Type* attribute.

```
syn Type Expression.Type();
eq Constant.Type() = getConstType();
eq Addition.Type() = (getOp1().Type()==Real||getOp2().Type()==Real)?
        Real:Int;
syn Value Expression.Value();
eq Constant.Value() = getConstValue();
eq Addition.Value() = Type()==Real?
        Real.sum(getOp1().Value(),getOp2().Value()):
        Int.sum(getOp1().Value(),getOp2().Value());
```

The JastAdd metacompiler generates a Java class for each node type and a method for each attribute within the class generated for the node type the attribute is associated with. The method body contains the code evaluating the attribute. Consequently, the generated AST classes *are* the AG evaluator.

3.3 Bridging the Gap

Considering the code generated for EMF-based metamodels and JastAdd specifications with respect to Section 2, an intuitive integration of semantics into EMF metamodels becomes evident. Assume, one uses JastAdd to implement the semantics of a metamodel. In such a case, the metamodel's node types (the `EClasses`) correspond to the classes generated for the AST specification, i.e., both have to share the same context-free structure. Furthermore, the semantics associated with non-containment references[6], derived properties and operations correspond to inherited and synthesized attributes of the AG. More precisely, each derived property's non-containment reference's or operation's EMF method corresponds to a method of the generated evaluator and as its implementation the one generated by JastAdd can be used.

Table 2 shows the proposed mapping of features for the integration of JastAdd with EMF metamodels while Figure 2 presents an example scenario for the well-known domain of a nested, block structured language and its name analysis semantics. Obviously, many AG features have one or more correspondences in the Ecore language such that an integration of both tools seems feasible. However, some features of EMF do not have a direct correspondence in the AG formalism. There is no direct correspondence for opposite references and the resolution of proxy objects. Opposite references are used to define pairs of references between `EClasses` as a kind of navigable bidirectional association. Although there is no similar language construct in JastAdd, the AG can be used easily to specify similar behaviour. The same holds for the proxy object resolution algorithm, which resolves external resources on demand: It would be easy to specify a non terminal attribute that loads a resource on demand since JastAdd's generated

[6] non-containment references in Ecore = non-composite references of section 2

JastAdd AG features	EMF Ecore features	General Metamodelling Language Concepts
AST node types	EClasses	Concepts
AST terminal children	EClass properties	Intrinsic values
AST non-terminal children	EClass containment references	Composite relationships
Synthesized attributes	EClass derived properties	Extrinsic values
	EClass operations	Operations
Inherited attributes	EClass derived properties	Extrinsic values
	EClass operations	Operations
Collection attributes	EClass derived properties*	Extrinsic values
	EClass non-containment references*	Non-composite relationships
Reference attributes	EClass non-containment references	Non-composite relationships
No matching AG formalism		
Can be simulated	Opposite references	Non-composite relationships
Can be simulated	Proxy objects	-
-	Dynamic EMF	-

Table 2. Proposed bidirectional mapping between JastAdd and EMF Ecore features w.r.t. the general metamodelling language concepts presented in Section 2.1. * means the feature has a cardinality greater one.

evaluators themselves are demand-driven. The only EMF feature that cannot be handled appropriately by JastAdd is its ability to dynamically instantiate metamodels without generating code (Dynamic EMF [27]).

Based on the mapping proposal, we implemented *JastEMF* - a tool that adapts JastAdd for EMF by merging the classes generated by both tools[7]. Figure 3 gives an overview of our tool's integration approach. The process starts from the JastAdd and Ecore input specifications designed w.r.t. Table 2. Thus, the AST defined by containment references in Ecore and the AST specified in JastAdd must be equal and the semantics of non-containments, operations and derived properties have to be specified by attributes and their equations in JastAdd. Code generation is controlled by *JastEMF*, which triggers the execution of EMF and JastAdd generating a class hierarchy according to the Ecore metamodel and the specified AST. To integrate both hierarchies we prepare the JastAdd classes to work with the EMF repository and merge the hierarchies using the *EMF merge tool*. The adaptation becomes necessary because of structural and behavioural conflicts and is implemented by automated refactoring operations (using Eclipse refactorings) and aspect weaving (using AspectJ [16]). These and other more complex conflicts that we discovered will be discussed in the next section.

[7] `http://www.jastemf.org`

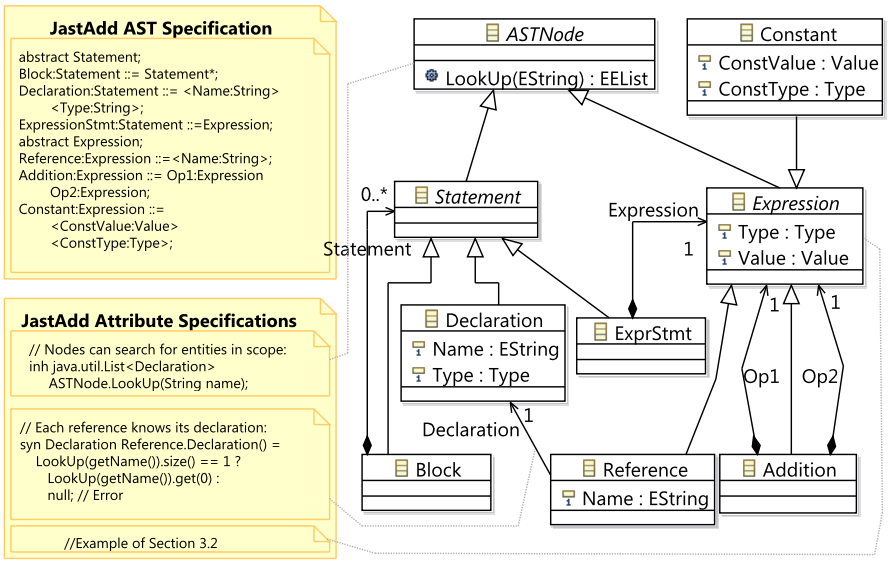


Fig. 2. JastAdd specification for an Ecore metamodel's name analysis semantics including the example of Section 3.2. Besides the concepts of the example, the metamodel on the right declares two additional semantic concepts: A non-composite relationship between `Reference` and `Declaration` and a *LookUp(EString)* operation associated with `ASTNode`. The specifications on the left define the semantic components using synthesised and inherited attributes.

3.4 Integration Conflicts

Even if the integration presented above is promising, several problems arise. We distinguish minor structural conflicts, complex behavioural, and usage conflicts. Minor structural conflicts are mainly caused by repairable API mismatches between the respective tools and concepts that are not explicitly included in one of the specification languages. Behavioural and usage conflicts stem from differences between the purposes of AGs and metamodels. In the following, we discuss both conflict groups and sketch our solutions.

Minor Structural Conflicts

- `EClass` properties with cardinality > 1 cannot be modeled in JastAdd's AST specification language. However, JastAdd terminals can have any valid Java type even collections. Since in EMF a special type is used to represent collection properties (the `EList`) an adaptation is necessary.
- ELists in EMF also represent containment references with cardinality > 1, whereas JastAdd has its own type to represent unbounded repetitions (Kleene closures). Hence, JastAdd's lists must be adapted to `EList`. Addi-

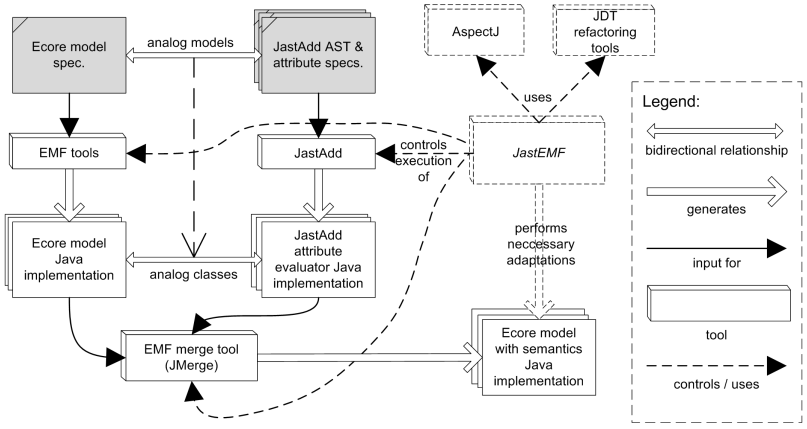


Fig. 3. *JastEMF* adaptation and integration process

tionally, bounded repetitions are not supported in JastAdd, though its lists can represent them.
- A non-null wrapper object (`Opt`) in JastAdd internally represents optional containment references. To adapt optional AST children, the merged classes have to access the wrapped object transparently[8].
- EMF allows modelling multiple inheritance while JastAdd does not. However, since Java does not support multiple inheritance, it is reduced to single inheritance and multiple interface inheritance. Since JastAdd supports inter-type declarations, it is no problem for an adaptation to extend AST classes with certain interfaces.
- All JastAdd nodes share a common supertype (`ASTNode`) which is generated separately for each AG evaluator. In EMF, all generated metamodel classes are of type `EObject`. To adapt the AST classes to EMF, it is sufficient to declare `EObject` as supertype of `ASTNode`.
- Metamodels in EMF can be arranged in packages (`EPackage`) which are reflected as Java packages in the generated class hierarchy. Since JastAdd does not support packages, refactorings become necessary to move the evaluator classes into the packages specified in the metamodel,.

Behavioural and Usage Conflicts

The main purpose of AGs in general is compiler construction for textual languages. Thus, AG evaluators expect complete input information, i.e., ASTs are complete and no interactive changes are expected. In contrast, metamodels have multiple use cases ranging from interactive graphical languages over domain specific languages and model transformations to editors and compilers for textual languages. Thus, some applications change models interactively with potentially

[8] The problem does not occur for optional properties and references.

incomplete or invalid intermediate states while others also expect complete models as an input. As a result, even if for many use cases the adapted JastAdd can immediately be applied to specify EMF-based metamodels semantics, interactive EMF applications require additional adaptations, e.g.:

- To support graphical editing with different views, models need to report changes to registered applications by emitting notification messages. JastEMF guarantees this behavior by weaving a notification aspect around the JastAdd methods for AST manipulations.
- Imperative attribute changes have to be supported, so the user can specify an attribute's value interactively. However, it has to be considered that imperative changes shadow attribute equations and may lead to inconsistent semantics. Especially, if attribute caching and the switching from automatic JastAdd evaluation to imperative specified values and back are considered, the necessary adaptations become complicated. In our current implementation we generate a simple Boolean flag for each attribute, that signals whether the attribute has to be computed by its equation or the user imperatively specified a value. An aspect guarantees, that always the attribute is accessed and the flag signals its value is given by the user, the given value is returned instead executing the attribute's equation. As soon as an attribute is imperatively specified, all attribute caches are cleared, to ensure correct values for attributes depending on it.

As an essence of the last two sections, the RAG approach seems compatible with metamodelling languages as long as the use case is similar. However, if the latter is not the case, behavioural conflicts reside on a conceptual level. There is a need for advanced attribute grammar tools that can cope with interactive usage scenarios. An efficient propagation of manual changes with respect to the AG, as presented in [23,20], is needed. Inconsistent intermediate states throughout interactive modelling activities have to be considered by the AG evaluator used to compute a model's semantic.

4 Related Work

Textual concrete syntax mapping tools like EMFText [15] combine existing parser generator technology with metamodelling technology [10]. They enable users to generate powerful text editors including features such as code completion and pretty printing. However, static semantics and code analysis still needs to be created manually. Consequently, such tools could immediately profit from our integration approach.

The Object Constraint Language (OCL) [12] is a standardised functional programming language for the specification of constraints, such as invariants and pre and post conditions, over UML artifacts. The EMF also provides an implementation for Ecore that is often used for the purpose of model validation

or to impose additional requirements on metamodels. In comparison to AGs, OCL is only convenient to define constraints that check for context-sensitive well fromedness based on certain values while AGs are additionally convenient to specify how such values can be computed. I.e. AGs do not only have a constraining character, but also a generative one.

Kermeta [21] provides an integrated environment for domain specific modeling language development based on the EMF. A language in Kermeta is developed by specifying its abstract syntax with an EMOF metamodel and static semantics with OCL constraints. Execution semantics can be implemented using a new imperative programming language. Abstract syntax, static semantics and execution semantics are developed in modules that can be combined using Kermeta's aspect language. The modularization concept supported by Kermeta's aspect language seems very similar to the aspect concept of JastAdd: They both support the separation of crosscutting semantic concerns. Additionally, Kermeta and JastEMF projects immediately benefit from EMF tooling in Eclipse.

However, we are sceptical about the Kermeta developers decision to design a new imperative programming language for execution semantics. Besides the additional effort for users to learn it, the vital question how well such programs can interact with other Java applications and can be integrated in heterogeneous project environments remains. We believe that the JastAdd developers decision to seamless integrate it with Java is very convenient, since users can switch to ordinary Java code at any time. Furthermore, we see no advantage of the interaction[9] between static semantics specified using OCL and imperative operations over static semantics which is specified using JastAdd and execution semantics specified by Java operations. Quite contrary, we believe that many execution semantic related problems occurring in the metamodelling world can be solved using JastAdd's rewrite capabilities[10].

The FUJABA approach [22] integrates UML class diagrams and graph rewriting to specify semantics of class operations. It provides *story driven modeling* as a visual language to define rewrite rules, which can be compared to UML activity diagrams. MOFLON [2] adapts FUJABA to support the Meta Object Faclility (MOF) as a modelling language. In general, graph-rewriting systems are harder to understand than AGs. Given a set of rewrite rules, it is complicated to foresee all possible consequences w.r.t. their application on start graphs. Rewriting results usually depend on the order of rule applications. To solve this problem, it is necessary to ensure that the rewrite system is confluent, which implies a lot of additional effort, not only for the proof of confluence, but also for the design of appropriate stratification rules. On the other hand, AGs

[9] Interaction here means that the computation of static semantics is initiated by execution semantics on demand.

[10] We specified a simple functional programming language in JastAdd, that supports Boolean, integer and vector arithmetic, control flow expressions, lexical scope and function closures and whose execution semantics is based on rewrites.

require a basic context-free structure or a spanning tree they are defined on whereas graph rewriting does not rely on such assumptions. Furthermore, RAGs can only add information to an AST but not remove them or even change its structure. However, there are AG concepts such as higher order attributes [28] (*non-terminal* attributes in JastAdd) and rewrite rules which improve in that direction.

Abstract State Machines and MOF. In [9], a framework for behavioural semantics of MOF-based languages is proposed. The approach is based on the Abstract State Machine (ASM) formalism which is mapped to the MOF. In contrast to our approach, the metamodelling language is extended. Furthermore, the approach has its focus on execution semantics and considers weaving of operations. On the contrary, JastEMF is better suited for static semantics and supports not only the specification of operations' semantics, but also of derived properties' and non-containment references' semantics. Similar to AGs, the ASM approach is theoretically backed and has been applied in multiple practical use cases [3].

Incremental AGs. The idea to apply AGs in interactive scenarios is not a new one. Especially for the development of interactive editors — as common in IDEs — incremental AG evaluation concepts have been investigated, which reduce the recomputation overhead in the presence of frequent context information changes [23,20]. Further developments in that direction for JastAdd also look promising, as shown by refactoring extensions for the extensible JastAddJ compiler [24,25]. Nevertheless, the support of interactive AST and attribute value changes, such that depending attributes are automatically updated, is still an open issue in JastAdd.

5 Conclusion and Outlook

In this paper, we discussed and motivated the integration of metamodelling approaches and attribute grammars to allow for the specification of metamodel semantics using a technique that is grounded in compiler theory. We showed in Section 2 that common metamodelling languages define a spanning tree for each model instance and how — based on that observation — RAGs can be used to specify metamodel semantics. Afterwards, we sketched a concrete integration for the JastAdd metacompiler and the Eclipse modelling Framework for which a prototype implementation exists. We believe that the presented overview supports research in either direction — a formalised mapping between metamodel semantics and RAGs or concrete tool implementations and example integrations. However, further effort with respect to interactive, incomplete or contracting user inputs is essential to establish the presented approach in interactive usage scenarios.

Acknowledgement

This research was funded by the *European Social Fond* in cooperation with the *Free State of Saxony* within the Ph.D. project *Innovative Werkzeugarchitekturen für die Erstellung von IT-Produktlinien im Bereich eingebetteter Software*and the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) within the project *HyperAdapt*.

We also like to thank Görel Hedin for her support and valuable comments, especially w.r.t. AGs and JastAdd.

References

1. The JastAdd Metacompiler. http://www.jastadd.org, 2009. website.
2. Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *LNCS*, pages 361–375. Springer, 2006.
3. E. Börger and B. Thalheim. Modeling workflows, interaction patterns, web services and business processes: The ASM-Based approach. In *Proc. of ABZ*, volume 5238 of *LNCS*, pages 24–38. Springer, 2008.
4. J. Craig Cleaveland and Robert C. Uzgalis. *Grammars for Programming Languages*. Programming Languages Series. Elsevier, 1977.
5. Torbjörn Ekman. *Extensible Compiler Construction*. PhD thesis, University of Lund, Sweden, 2006.
6. Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.
7. Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 85–98. ACM, 1986.
8. The Eclipse Foundation. EMF-based implementation of the UML2 metamodel. http://www.eclipse.org/modeling/mdt/?project=uml2, 2009. website.
9. Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4):415–454, 2009.
10. Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proc. of ECMDA-FA*, volume 5095 of *LNCS*. Springer, 2008.
11. Object Management Group. Meta Object Facility (Version 2.0). http://www.omg.org/spec/MOF/2.0/, 2006. website.
12. Object Management Group. Object Constraint Language (Version 2.0). http://www.omg.org/spec/OCL/2.0/, 2006. website.
13. Görel Hedin. An Object-Oriented Notation for Attribute Grammars. In *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 329–345. Cambridge University Press, 1989.
14. Görel Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3), 2000.
15. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 114–129. Springer, 2009.

16. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer, 2001.
17. Donald E. Knuth. Semantics of Context-Free Languages. *Theory of Computing Systems*, 2(2):127–145, 1968.
18. Donald E. Knuth. Semantics of Context-Free Languages: Correction. *Theory of Computing Systems*, 5(2):95–96, 1971.
19. Cornelis H. A. Koster. Affix Grammars for Programming Languages. In *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 358–373. Springer, 1991.
20. William H. Maddox III. *Incremental Static Semantic Analysis.* PhD thesis, University of California at Berkeley, 1997.
21. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.
22. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745. ACM, 2000.
23. Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):449–477, 1983.
24. Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 277–294. ACM, 2008.
25. Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping Stones over the Refactoring Rubicon – Lightweight Language Extensions to Easily Realise Refactorings. In *ECOOP '09: 23rd European Conference on Object-Oriented Programming*, volume 5653 of *LNCS*, pages 369–393. Springer, 2009.
26. Bran Selic. The Theory and Practice of Modeling Language Design for Model-Based Software Engineering – A Personal Perspective. In *GTTSE '09: Proceedings of the 3th Summer School on Generative and Transformational Techniques in Software Engineering*, LNCS. Springer, 2010. to appear.
27. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF – Eclipse Modeling Framework.* Addison-Wesley, 2 edition, 2008.
28. Harald H. Vogt, Doaitse Swierstra, and Matthijs F. Kuiper. Higher Order Attribute Grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145. ACM, 1989.