Reference Attribute Grammars for Metamodel Semantics

Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann

Institut für Software- und Multimediatechnik
Technische Universität Dresden
Dresden, Germany
christoff.buerger|sven.karol|c.wende|uwe.assmann@tu-dresden.de

Abstract. While current metamodelling languages are well-suited for the structural definition of abstract syntax and metamodelling platforms like the Eclipse Modelling Framework (EMF) provide various means for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured. Therefore, we propose the application of reference attribute grammars (RAGs) to alleviate the lack of support for formal semantics specification in metamodelling. We contribute the conceptual foundations to integrate metamodelling languages and RAGs, and present *JastEMF* — a tool for the specification of EMF metamodel semantics using JastAdd RAGs. The presented approach is exemplified by an integrated metamodelling example. Its advantages, disadvantages and limitations are discussed and related to metamodelling, attribute grammars (AGs) and other approaches for metamodel semantics.

1 Introduction

Metamodelling is a vital activity for Model-Driven Software Development (MDSD). It covers the definition of structures to represent abstract syntax models, the specification of a concrete syntax, and the specification of the meaning of models [1]. While infrastructures like the Eclipse Modelling Framework (EMF) [2] provide means for the specification of abstract syntax and various associated tools for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured [1].

In this paper we propose the application of RAGs [3] — a well-investigated extension of Knuth's classic AGs [4] — to alleviate the lack of support for formal semantics specification in metamodelling. They enable (1) the specification of semantics on tree and graph-based abstract syntax structures with unique spanning trees, (2) completeness and consistency checks of semantic specifications, and (3) the generation of an implementation of semantics specifications.

The contributions of this paper are as follows: The next section discusses common concerns in the specification of metamodels, identifies key capabilities we target with *semantics-integrated metamodelling* and introduces a motivating example. In Section 3

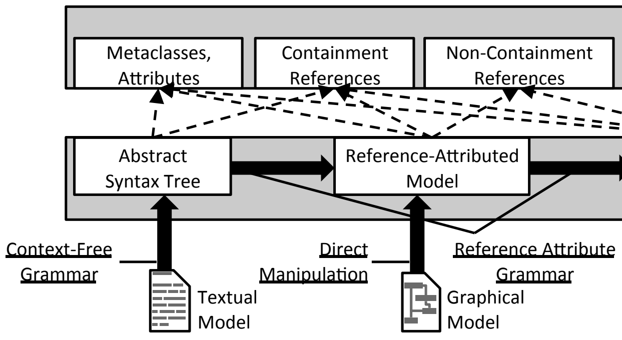


Fig. 1. Transformations in Metamodelling.

we sketch general foundations for the application of RAGs for metamodel semantics — our semantics-integrated metamodelling approach. In Section 4 we demonstrate the feasibility of this idea by presenting *JastEMF*, a tool that integrates the Ecore metamodelling language and the JastAdd [5] attribute grammar system. In Section 5 we describe the application of our semantics-integrated metamodelling approach to implement the SiPLE-Statemachines language introduced in Section 2. In Section 6 we evaluate JastEMF against the key capabilities identified for semantics-integrated metamodelling based on our experiences with the SiPLE-Statemachines case study. Finally, we discuss related work in Section 7 and conclude our contribution in Section 8.

2 Motivation for Semantics-Integrated Metamodelling

This section motivates our idea of semantics-integrating metamodelling with RAGs by identifying methodical gaps to achieve common objectives in metamodelling. Afterwards, we introduce a set of key capabilities we target with *semantics-integrated metamodelling* and introduce SiPLE-Statemachines, an exemplary metamodel project that serves as a continuous example throughout this paper.

2.1 Metamodelling: Objectives, Transformations, Specifications

Metamodelling has the objective to specify the implementation of a modelling language. Such implementation should provide means to transform programs (i.e., models) starting from (1) their textual representation to (2) their abstract syntax representation and finally (3) representations of their static and execution semantics [1]. In the metamodelling world, all these representations and transformations are related to the language's metamodel, which typically *declares* the data structures that are used for representing language constructs in abstract syntax models and is the interface for the *specification* of concrete syntax and semantics. As depicted in Figure 1, each transformation's input and output model is built using specific kinds of constructs of the language metamodel. A first transformation — typically specified using regular expressions and context-free grammars — derives an abstract syntax tree (AST) from textual symbols. The data structure required to represent the AST is solely declared by the `Metaclasses`, `Attributes` and `Containment References` in the metamodel. In a second transformation, the structures that are declared as `Non-Containment References` (e.g., connecting variable usage with variable declarations) need to be derived, resulting in a reference-attributed model — i.e., the abstract syntax tree with a superimposed reference graph. Graphical editors often directly operate on such reference-attributed models (cf. Figure 1) and rely on a direct manipulation of non-containment references. A last transformation performs semantics evaluations to derive values for `Derived Attributes` and executes `Operations` declared in the metamodel. Note that the reference-attributed model and the full-attributed model still contain the abstract syntax tree as their *spanning tree*.

All the above mentioned transformations are important artefacts of a language and, thus, motivate a formal definition. However, formal approaches for the specification of static or execution semantics are not yet established in the metamodelling world. As depicted in Figure 1 we aim at closing this gap by the application of RAGs for the specification of metamodel semantics.

2.2 Capabilities of Semantics-Integrated Metamodelling

By applying RAGs to achieve *semantics-integrated metamodelling* we expect to combine the benefits of metamodelling frameworks and attribute grammars. However, to our experience, integration means not only combination of benefits but often also a compromise of the technical and methodical capabilities of the individual approaches. We therefore collected a number of technical key capabilities to be contributed by each individual approach that afterwards will be used to evaluate our integrated solution.

Metamodelling frameworks (e.g., the EMF) are built around a metamodelling language (e.g., Ecore) and typically provide tools for the specification and implementation of modelling languages and their tooling. In particular they provide:

MM 1: Metamodelling Abstraction: Metamodelling language that provides a dedicated abstraction to specify language metamodels.
MM 2: Metamodelling Consistency: Tools to check the structural completeness and consistency of metamodel specifications.
MM 3: Metamodel Implementation Generators: Generators to derive implementations from metamodel specifications.
MM 4: Metamodel/Model Compatibility: A common repository and representation that enables integration of modelling languages and models.
MM 5: Tooling Compatibility: Common platform for tool integration/application:
- **MM 5.1: Model-to-Model Transformation Tools:** E.g., ATL [6] or XTend [7].
- **MM 5.2: Model-to-Text Transformation Tools:** Code generators and model-driven template languages like Mofscript [8] or XPand [7].
- **MM 5.3: Text-to-Model Transformation Tools:** Parser generators for models as EMFText [9], Monticore [10] or XText [11].
- **MM 5.4: Generic Model Editors:** Generic tools to access and edit models as the Generic EMF editor [2] or Exeed [12].

MM 5.5: Tooling Generators: Tooling to specify and generate textual (EMFText, Monticore, XText) or graphical (GMF [2], EuGENia[1]) model editors.

Attribute grammar systems typically provide means for the specification of languages' abstract syntax and semantics and tools to derive an implementation from such specifications. In particular they provide:

AG 1: Semantics Abstraction: Well-investigated, declarative abstraction for formal semantics specifications.
AG 2: Semantics Consistency: Tooling to check the structural completeness and consistency of semantics specifications.
AG 3: Semantics Generators: Generators to derive an implementation (i.e., evaluator) from semantics specifications.
AG 4: Semantics Modularity: Modular, extensible semantics specifications [13].

2.3 SiPLE-Statemachines: A Typical Modelling Language

To exemplify and evaluate semantics-integrated metamodelling we will use a typical modelling scenario. It is built upon a Simple imperative Programming Language Example (SiPLE) and a statemachine language which are combined to support the modelling of executable statemachines.

SiPLE's main features are boolean, integer, and real arithmetics, nested scopes, nested procedure declarations, recursion, while loops and conditionals. All these features of SiPLE have the intuitive semantics familiar from imperative programming languages. Listing 1.1 shows a basic SiPLE program that asks the user for a number, computes its Fibonacci value and prints it.

Listing 1.1. Fibonacci Numbers in SiPLE

```
Procedure main() Begin
 Procedure fibonacci(Var n:Integer) : Integer Begin
  If n = 0 Or n = 1 Then Return 1; Fi;
  Return fibonacci(n-2) + fibonacci(n-1);
 End;

 Var n:Integer;
 Read n;
 Write fibonacci(n);
End;
```

Statemachines describe the behaviour of systems using a state-based abstraction [14]. In contrast to the textual syntax of SiPLE, they are typically modelled using a graphical notation. The exemplary statemachine depicted in Fig. 2 describes the behaviour of a phone. It uses concepts like states (e.g., `Dialing`), transitions (e.g., `incoming call`), guard conditions and actions. With transitions the phone reacts on particular events from the environment by changing states, e.g., `incoming call` where the phone changes from `Waiting` to `Ringing`. Guards and actions enable a more fine-grained specification of boolean conditions and imperative behaviour, respectively. Therefore, we want to combine the statemachine language and SiPLE to

[1] http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/

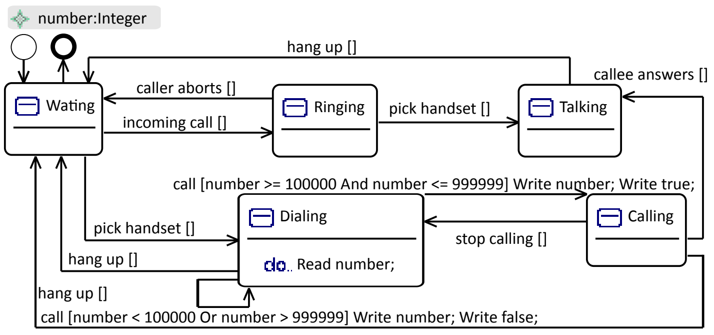


Fig. 2. Phone example modelled in a generated GMF editor for SiPLE-Statemachine.

SiPLE-Statemachine. Consider for instance the state `Dialing` where an entry action is used to read a number from the user (`Read number;`). SiPLE action statements and boolean guard expressions can also be associated to transitions. For instance, when the phone is in the `Dialing` state and receives a `call` event, it checks the entered number using a boolean SiPLE expression and changes its state in accordance. While walking the according transition it writes the dialed or rejected number to the standard output (`Write number; Write true;` or `Write number; Write false;`).

In this paper we will demonstrate and evaluate the integration of metamodelling and RAGs by implementing SiPLE-Statemachines. Relying on the key capabilities identified in Section 2.2 we plan to provide:

– A metamodel (MM 1, MM 2, MM 3), textual concrete syntax (MM 5.3, MM 5.5) and attribute-grammar semantics (AG 1, AG 2, AG 3) for SiPLE.
– An interpreter that implements an execution semantics for SiPLE (MM 5).
– The SiPLE-Statemachine metamodel (MM 1, MM 2, MM 3) composed from the two metamodels of SiPLE and Statemachine (MM 4).
– A generated graphical editor for SiPLE-Statemachine (MM 5.5).
– An attribute-grammar semantics for SiPLE-Statemachine (AG 1, AG 2, AG 3, AG 4), e.g., for mapping transition labels to states and to reason about state reachability.
– An execution semantics for SiPLE-Statemachine by implementing a model-to-text transformation (MM 5.2) for statemachines to plain SiPLE code.

3 Foundations of Attribute Grammars for Metamodel Semantics

Because RAGs rely on a specific representation of abstract syntax, their application in metamodelling requires an integration with metamodel constructs. In this section we prepare such integration by investigating RAGs' specific syntax requirements, clarifying what kind of semantics they can specify and which kind of model information these semantics represent, and finally showing that most metamodelling languages indeed satisfy RAGs' syntax requirements.

3.1 Reference Attribute Grammars and Metamodel Semantics

RAGs are used to specify semantics for tree structures that are usually specified using a context-free grammar (CFG). Given a tree the RAG annotates it with values and imposes a graph on it representing the language's semantics. Because we like to use RAGs for metamodel semantics, we must identify metamodel constructs that induce such tree-structure in model instances. We like to use our approach not only for certain metamodels, but rather for any metamodel developed in a metamodelling language. Thus, the separation of metamodel constructs into tree- and graph-inducing structures — into syntax and semantics — should be metamodelling language inherent, i.e. be implied by the meta-metamodel[2]. For this purpose, we investigate common metamodelling languages' concepts and distinguish them into syntactic (tree structure *defining*) and semantic (graph structure *declaring*) ones. Both sets will be disjunct. Given this separation RAGs are indeed appropriate to specify metamodel semantics in the sense that they can be used to specify the transformation of abstract syntax trees to reference attributed graphs and full-attributed graphs (cf. Figure 1). RAGs can be used for any kind of static model semantics like model checking and analysis. Definition 3.1 summarises the foundations of integrating RAGs and metamodels:

Definition 3.1 (Metamodel Semantics): Let Ω be a metamodel and E_Ω the finite set of its elements. Let E_{syn_Ω} and E_{sem_Ω} be disjunct subsets of E_Ω, whereas $E_\Omega = E_{syn_\Omega} \cup E_{sem_\Omega}$. Let E_ω be the set of entities of a model instance $\omega \in \Omega$. Since $\omega \in \Omega$, all entities $e \in E_\omega$ have a type $t_e \in E_\Omega$. Let S_Ω be a function that defines for all $\omega \in \Omega$ for each entity $e \in E_\omega$ with $t_e \in E_{sem_\Omega}$ the value of e. We call S_Ω a metamodel semantic for Ω. Iff E_{syn_Ω} specifies a spanning tree for each $\omega \in \Omega$, S_Ω can be specified with a RAG.

The metamodel semantics S_Ω can depend on any metamodel element $me \in E_\Omega$. They even can depend on themselves, in which case they are only well-defined if there exists a fix-point. Thus, different model instances can only have different semantics, if the semantics depend on syntactic elements $me \in E_{syn_\Omega}$. Colloquially explained, a model's semantics (i.e., all entities $\{e | e \in E_\omega \wedge t_e \in E_{sem_\Omega}\}$) depend on its structure (i.e., all entities $\{e | e \in E_\omega \wedge t_e \in E_{syn_\Omega}\}$).

What remains to show is, which metamodelling concepts belong to E_{syn_Ω} and E_{sem_Ω} and that indeed E_{syn_Ω} specifies a tree structure.

3.2 Common Metamodelling Languages and Abstract Syntax Trees

Most metamodelling languages support (1) *metaclasses* consisting of (2) *non-derived* and (3) *derived attributes* and (4) *operations*. Metaclasses can be related to each other by (5) *containment* and (6) *non-containment relationships*. Non-derived attribute values represent AST terminals in models and, thus, are in E_{syn_Ω}. Containment references model that instances of a metaclass C_1 consist of instances of a metaclass C_2.

[2] Most metamodelling languages are specified in themselves, such that it is appropriate just to talk about metamodels in the following.

The contained C_2 instances are an inextricable, structural part of the C_1 instances. The relationship between C_1 and C_2 is a composite and iff an instance $e_2 \in C_2$ is a composite of an instance $e_1 \in C_1$, e_1 cannot be a composite of e_2. Thus, containment relationships specify tree structures. They are in E_{syn_Ω}. Derived attributes and side-effect free operations model the values that can be calculated from other values of a given model. Non-containment relationships model arbitrary references between metaclasses. Thus, derived attributes, side-effect free operations, and non-containment relationships are in E_{sem_Ω}. Operations with side-effects can model either, extensions of models derived from existing model information or arbitrary model manipulations. Derived model extensions are in E_{sem_Ω}, because they can be considered to be part of the graph imposed by semantics (w.r.t. AGs such derived model extensions are higher-order attributes [15]). Operations that represent arbitrary model manipulations (e.g., to delete model elements or imperatively add new elements) cannot be handled by our definition of metamodel semantics.

3.3 Graphs and (Partial) Reference-Attributed Models

In the domain of modelling, often reference-attributed models (cf. Section 2.1 and Fig. 1) are the starting point for semantic evaluations. A typical scenario are models developed in graphical editors. Of course, it is no problem for a RAG-based metamodel semantic S_Ω if elements of E_{sem_Ω} of a model instance have a predefined value — i.e., if instead of a tree the semantic evaluation starts from a graph with a unique spanning tree.

Throughout semantic evaluation, a RAG evaluator can use such predefined values and simply ignore their specified semantics. If for every model instance and all its occurences the value of a non-containment reference is always predefined, the specification of its semantics can even be omitted. Thus, (partial) reference-attributed models do not influence the applicability of our RAG approach for metamodel semantics.

4 JastEMF: An Exemplary Attribute Grammar and Metamodelling Language Integration

In this section, we discuss the integration of an exemplary metamodelling framework (EMF [2]) and RAG system (JastAdd [5]). We shortly introduce both approaches and then discuss the details of their integration in *JastEMF*.

4.1 The Eclipse Modelling Framework

The EMF is a common metamodelling infrastructure for the Eclipse platform providing metamodel development and implementation tools based on the metamodelling language Ecore [2]. EMF contributes tools to edit Ecore metamodel specifications, check their consistency and generate a Java-based implementation of the metamodel specifications. The framework is used for the implementation of a plethora of modelling languages[3], and is an important integration platform for various modelling tools.

[3] http://www.emn.fr/z-info/atlanmod/index.php/Ecore

For the definition of concrete syntax the EMF is complemented by various tools to specify a concrete syntax in relation to a metamodel. Editor generators that are tightly integrated with EMF like EMFText [9] or XText [11] enable the declarative specification of context-free grammars to define parsers, printers, and advanced textual editors for models. There are also tools to realise a graphical (diagrammatic) model syntax, e.g., the Graphical Modelling Framework (GMF) [2].

For the specification of semantics, the EMF mainly relies on Java source code that evaluates derived attributes or implements operations declared in the metamodel. Besides the application of the Object Constraint Language (OCL) [16] or model-transformations, we are not aware of formal, mature techniques to specify static and execution semantics in EMF. For a further discussion of approaches for metamodel semantics we refer to Section 7.

4.2 The JastAdd Metacompiler

JastAdd [5, 13] is an object-oriented compiler generation system. It allows to generate a Java implementation of a demand-driven semantics evaluator from a given AG. Besides the basic attribute grammar concepts [4], JastAdd supports advanced AG extensions such as reference [3] (RAGs) and circular attributes [17].

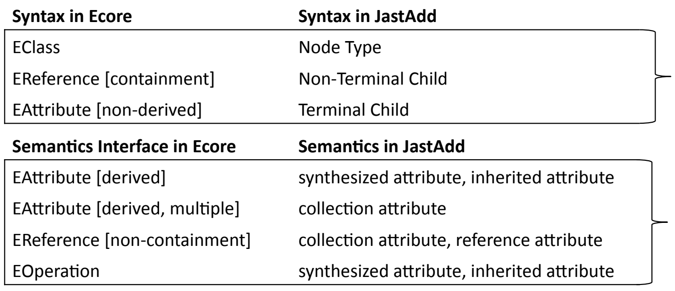
JastAdd has two specification languages. One to specify abstract syntax and another to specify an attribution (i.e. semantics). Abstract syntax specifications consist of node type declarations (non-terminals) and their child nodes (arbitrary list of terminals and non-terminals). Language semantics is usually specified within several modules containing attribute definitions and attribute equations that are associated with node types of the abstract syntax.

Given a set of AST and attribute specifications the JastAdd compiler generates a Java class for each node type, accessors for the node's children nodes, and methods for each attribute defined for the node type. The code required for attributes' evaluation is generated into their method bodies. Consequently, evaluation of semantics can be triggered by accessing the corresponding methods.

4.3 Integrating EMF and JastAdd

Because both EMF and JastAdd provide code generation for Java, they are well-suited to explore semantics-integrated metamodelling. For their practical integration it is required (1) to merge the Java classes that represent a language's abstract syntax in EMF and in JastAdd and (2) to apply the generated attribute evaluator to compute EMF models' semantics.

Based on the integration foundations presented in Section 3 we derived a mapping of elements in the Ecore metamodel and specification concepts used by JastAdd. The concrete mappings depicted in Figure 3 are grouped in two sets. The first set contains elements related only to model *syntax* (E_{syn_Ω}). The second set contains the elements related to model *semantics* (E_{sem_Ω}). In the second set constructs of the Ecore metamodel are typically used to declare the *semantics interface* of specific elements while the corresponding JastAdd construct specifies the element's semantics. Depending on its actual syntax and semantics, multiple mappings are possible.



Syntax in Ecore	Syntax in JastAdd
EClass	Node Type
EReference [containment]	Non-Terminal Child
EAttribute [non-derived]	Terminal Child

Semantics Interface in Ecore	Semantics in JastAdd
EAttribute [derived]	synthesized attribute, inherited attribute
EAttribute [derived, multiple]	collection attribute
EReference [non-containment]	collection attribute, reference attribute
EOperation	synthesized attribute, inherited attribute

Fig. 3. Integrating Ecore and JastAdd.

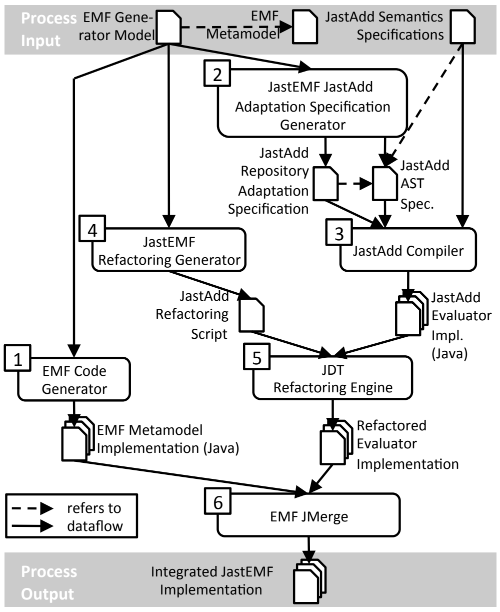
In general, derived properties, non-containment references and operations that are side-effect free are considered to be static semantics, whereas their semantics can be specified using synthesized or inherited attributes (reference attributes in the case of a non-containment reference). If the cardinality of a derived property or non-containment reference is greater than one, often collection attributes [18, 19] are much more convenient than ordinary attributes, since they permit to collect remotely located AST nodes w.r.t. conditions and reference attributes. However, since JastAdd attributes can have any valid Java type, it is also possible to specify ordinary attributes that represent collections. Operations with side-effects should not be realized by attributes, but rather by ordinary Java methods specified within JastAdd attribute specifications. Such intertype declarations are woven by JastAdd as known from aspect weaving tools like AspectJ [20].

JastEMF's Integration Process To realise the integration of EMF and JastAdd, we implemented *JastEMF*[4]. Given an Ecore metamodel with in JastAdd specified semantics — a so called *JastEMF integration project* — JastEMF can be executed to generate an integrated language implementation, i.e. an EMF metamodel implementation with integrated JastAdd semantics. *Integration projects* must provide the following artefacts:

– An Ecore metamodel declaring the language's abstract syntax.
– An Ecore generator model configurating EMF and JastEMF code generation.
– A set of JastAdd attribute specifications defining the language's semantics that satisfy the mappings defined for concepts in E_{sem_Ω} (cf. Fig. 3).

Based on these artefacts, JastEMF's generation process (cf. Figure 4) reuses the generators for JastAdd and EMF and merges the generated Java classes in accordance to the introduced mapping. First, the process uses the EMF Generator Model, which is fed to the `(1) EMF Code Generator` to generate an EMF Metamodel Implementation and the `(2) JastEMF JastAdd Adaptation Specification Generator` to derive a JastAdd AST Specification and a JastAdd Repository Adaptation Specification. The Repository Adaptation Specification contributes attribute specifications

[4] `www.jastemf.org`



Process Input
EMF Generator Model
EMF Metamodel
JastAdd Semantics Specifications
2
JastEMF JastAdd Adaptation Specification Generator
JastAdd Repository Adaptation Specification
JastAdd AST Spec.
4
JastEMF Refactoring Generator
3
JastAdd Compiler
JastAdd Refactoring Script
JastAdd Evaluator Impl. (Java)
1
EMF Code Generator
5
JDT Refactoring Engine
EMF Metamodel Implementation (Java)
Refactored Evaluator Implementation
refers to
dataflow
6
EMF JMerge
Process Output
Integrated JastEMF Implementation

Fig. 4. JastEMF's Generation Process.

that adapt the JastAdd Evaluator Implementation to use the EMF repository instead of its own internal repository. As a second input the process requires the JastAdd Semantics Specifications. The JastAdd AST Specification, the JastAdd Repository Adaptation Specification and the JastAdd Semantics Specifications are used by the `(3) JastAdd Compiler` to generate a JastAdd Evaluator Implementation. To integrate this Evaluator with the Metamodel Implementation it has to be refactored to incorporate metamodel naming conventions and package structures. Therefore, the `(4) JastEMF Refactoring Generator` derives a JDT[5] refactoring script from the metamodel and applies it `(5)`. For an overview of the refactorings applied we refer to [21]. Finally, the Refactored Evaluator Implementation is merged with the EMF Metamodel Implementation using `(6) EMF JMerge`. This last step results in a metamodel implementation with tightly integrated semantics, where semantic declarations from the EMF metamodel are combined with their attribute-based specifications defined in the JastAdd semantics.

With JastEMF the complexity of this integration process is completely hidden for developers. Also, developers can work with EMF and JastAdd as usual.

5 SiPLE-Statemachines Case Study

In the following, we present the application of semantics-integrated metamodelling for implementing SiPLE-Statemachine.

[5] http://www.eclipse.org/jdt/

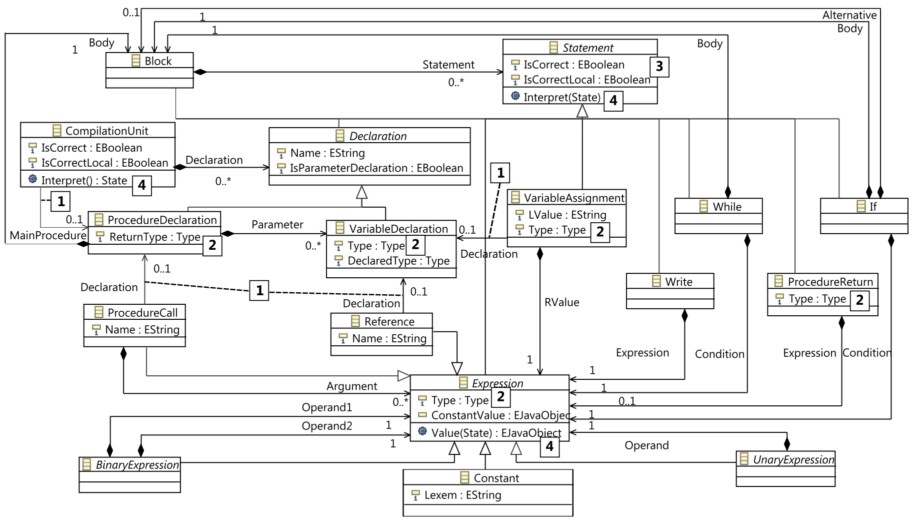


Fig. 5. SiPLE Metamodel.

Therefore, we discuss: (1) the application of EMF Ecore for specifying and integrating SiPLE and SiPLE-Statemachine abstract syntax, (2) the application of JastAdd for the specification of SiPLE and SiPLE-Statemachine static and execution semantics, and (3) the application of JastEMF to generate an integrated SiPLE-Statemachine implementation.

5.1 Modelling Abstract Syntax with EMF

SiPLE Abstract Syntax The SiPLE metamodel is presented in Figure 5. A `CompilationUnit` consists of `Declarations`, which can be `VariableDeclarations` declaring a variable's name and type or `ProcedureDeclarations` declaring a procedure. Each procedure has a name, a return type, a list of parameters and a body that is a `Block`. Each `Block` consists of a `Statement` sequence. In SiPLE, nearly everything is a `Statement`, such as `While` loops, `If` conditionals, `Expressions`, `Declarations` and `VariableAssignments`. Expressions can be `BinaryExpressions` or `UnaryExpressions`, whose concrete sub-classes, e.g., `Addition` and `Not`, are not presented in the figure. Furthermore, there are primitive expressions such as `Constants`, `References` and `ProcedureCalls`.

The metamodel specifies a spanning tree (containment references, non-derived attributes) enriched with semantics interfaces (non-containment references, derived attributes, operations). For a better understanding, we assigned numbers to different parts of the semantics interfaces declared in the metamodel. Parts that are to be computed by name analysis are marked with $\boxed{1}$, e.g., each `VariableAssignment` and each `ProcedureCall` in a well-formed program has a reference pointing to their respective declaration. Parts depending on type analysis are labeled by $\boxed{2}$, e.g., the derived attribute `Type` represents the actual type of an `Expression`. $\boxed{3}$ marked parts belong

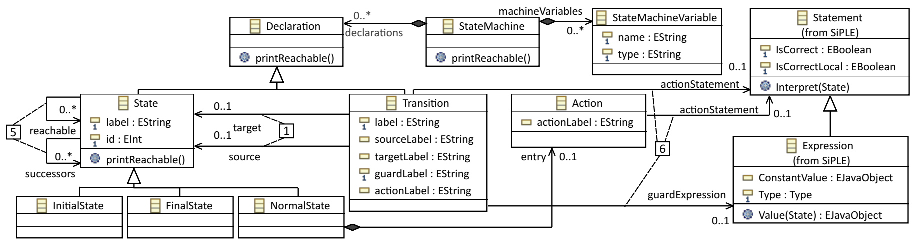


Fig. 6. SiPLE-Statemachine Metamodel.

to the constraint checking realization, which currently consists of the `IsCorrect` and `IsCorrectLocal` attributes. Parts labeled with $\boxed{4}$ (e.g., `Interpret()`) declare the execution semantics interface for both runtime evaluation and constant folding.

SiPLE-Statemachine Abstract Syntax The SiPLE-Statemachine metamodel is presented in Figure 6. A `StateMachine` consists of a set of `State` and `Transition` `Declarations`. Each `Transition` has a label representing an event that triggers the `Transition`. Furthermore, guard `Expressions` allow to specify boolean SiPLE expressions as additional conditions for transition triggering and SiPLE `Statements` allow to annotate actions to states and transitions that are executed when a state is entered or a transition is triggered respectively. Each `Transition` refers to a source and a target `State`.

The metamodel's semantics are an extension of the semantics used for the JastAdd statemachine tutorial in [22]. Again, we assigned numbers to parts of the semantics interface. As above, $\boxed{1}$ marks parts belonging to name analysis. In SiPLE-Statemachine name analysis is used to compute the actual source and target states from the corresponding labels of a `Transition` object. Further semantics analysis that computes additional information such as the successor relation and transitive closure of all states reachable from a given state are labeled by $\boxed{5}$. Parts that are used to parse textual action and guard labels to appropriate SiPLE fragments are marked with $\boxed{6}$.

5.2 Specifying Semantics with JastAdd

SiPLE Semantics There are four semantic concerns that completely specify SiPLE's static and execution semantics. With JastAdd, each concern can be specified as an *aspect*. A *core specification* is used to *declare* each concern's semantics, i.e., to *declare* all attributes (cf. Listing 1.2). The actual attribute *definitions* (the equations) reside in separate JastAdd specifications[6].

Listing 1.2. Excerpt from the SiPLE Core Specification

```
aspect NameAnalysis { // [1] in Figure 5
```

[6] All specifications can be found at www.jastemf.org.

```
  // Procedure name space:
  inh Collection<ProcedureDeclaration> ASTNode.LookUpPDecl(String name);
  syn ProcedureDeclaration ProcedureCall.Declaration();
  syn ProcedureDeclaration CompilationUnit.MainProcedure();

  // Variable name space:
  inh Collection<VariableDeclaration> ASTNode.LookUpVDecl(String name);
  syn VariableDeclaration Reference.Declaration();
  syn VariableDeclaration VariableAssignment.Declaration();
}
aspect TypeAnalysis { // [2] in Figure 5
  syn Type VariableDeclaration.Type();
  syn Type VariableAssignment.Type();
  syn Type ProcedureReturn.Type();
  syn Type Expression.Type();
}
aspect ConstraintChecking { // [3] in Figure 5
  syn boolean ASTNode.IsCorrect();
  syn boolean ASTNode.IsCorrectLocal();
}
aspect Interpretation { // [4] in Figure 5
  public abstract Object Expression.Value(State vm) throws InterpretationException;
  public abstract void Statement.Interpret(State vm) throws InterpretationException;
  syn State CompilationUnit.Interpret();
}
```

NameAnalysis SiPLE uses separate block-structured namespaces for variable and procedure declarations. Although there is a single global scope in each `Compilation-Unit`, each block introduces a new private scope, shadowing declarations in the outside scope. No explicit symbol tables are required to resolve visible declarations — the AST is the symbol table.

TypeAnalysis SiPLE is a statically, strongly typed language. For each kind of expression its type is computed from the types of its arguments, e.g., if an addition has a `Real` number argument and an `Integer` argument the computed type will be `Real`. Types are statically computed for arithmetic operations, assignments, conditionals (`If`, `While`), procedure calls and procedure returns.

ConstraintChecking Each language construct of a SiPLE program can be statically checked for *local* correctness, i.e., whether the node representing the construct satisfies all *its* context-sensitive language constraints or not. Of course, these checks are usually just simple constraints specified based on SiPLE's name and type analysis like "an `If` condition's type must be boolean" or "each reference must be declared". If all nodes of a (sub)tree — i.e., a program (fragment) — are local correct, the (sub)tree is correct.

Interpretation SiPLE's execution semantics is also specified using JastAdd. We applied JastAdd's ability to use Java method bodies for attribute specifications. This allows for a seamless integration of a Java implementation of the operational semantics and the declarative, RAG-based static semantics analysis. The interpretation is triggered with a call to a `CompilationUnit`'s `Interpret()` operation that initialises a state object representing a stack for procedure frames and traverses the program's statements by calling their `Interpret(State)` operation.

SiPLE-Statemachine Semantics The SiPLE-Statemachine semantics is mainly specified in three JastAdd aspects which are shown in Listing 1.3. The original source comes from [22]. To integrate SiPLE, we additionally introduced a `SiPLEComposition` aspect.

Listing 1.3. Excerpt from the SiPLE-Statemachine Core Specification

```
aspect NameAnalysis { // [1] in Figure 6
    syn lazy State Transition.source();
    syn lazy State Transition.target();
    inh State Declaration.lookup(String label);
    syn State Declaration.localLookup(String label);
}

aspect Reachability{ // [5] in Figure 6
  syn EList State.successors() circular [...];
  syn EList State.reachable() circular [...];
}
aspect SiPLEComposition { // [6] in Figure 6
  public Statement Action.getActionStatement();
  public Expression Transition.getGuardExpression();
  public Statement Transition.getActionStatement();
}
```

NameAnalysis In SiPLE-Statemachine, name analysis maps textual labels in transitions to states. Since states are not block-structured, all declarations of the statemachine are traversed and each state label is compared to the looked up label. Note that a graphical editor may set a transition's source and target state directly.
Reachability The synthesized `successor` attribute computes a state's direct successor relation from the set of declared transitions. In contrast to the original example, we declared the attribute to be an `EList` to achieve better graphical editor support and as circular because of editor notifications issues. Based on the successor relation, the `reachable` attribute computes a state's transitive closure, which can be displayed on demand in a graphical editor we generated for SiPLE-Statemachines.
SipleComposition This helper aspect uses the SiPLE parser to parse and embed SiPLE `Expressions` and SiPLE `Statements` for guard and action labels. Actually, these parts belong to SiPLE and are integrated into SiPLE-Statemachine (cf. 6). However, since JastAdd does not support packages and always generates AST classes for the given AST specifications instead of reusing classes from existing packages as supported by the EMF, we had to model them as attributes.

5.3 Integration with Further Metamodelling Tools

To prepare the evaluation of JastEMF's integration approach we applied several metamodelling tools for further implementation tasks. As our focus is about the specification of semantics for metamodels using RAGs, we shortly describe their purpose but refer to respective publications for details:

- To generate a parser and an advanced text editor for SiPLE we use EMFText [9]. Amongst others, the generated editor supports syntax highlighting, code completion, outline and a properties view.
- To provide a graphical syntax for SiPLE-Statemachine we used EuGENia[7], a tool that processes specific metaclass annotations to generate based on their information (node or transition, shape, style, color, etc.) an according set of GMF [2] specifications. From such specifications the GMF framework then generates a well-integrated, powerful graphical editor for the SiPLE-Statemachine language (cf. Figure 2).

[7] http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/

– To make SiPLE-Statemachine executable, we use an XPand [7] template to generate plain SiPLE code, which can be executed by a call to its `CompilationUnit`'s `Interpret` operation.

6 Evaluation

In this section we evaluate JastEMF based on our experience with the SiPLE-Statemachines case study and w.r.t. the semantics-integrated metamodelling capabilities presented in section 2.2. Afterwards, we discuss limitations of our approach and further experiences in applying JastEMF that motivate deeper investigation and future work.

6.1 Evaluating the Capabilities of Integrated Metamodelling

The JastEMF integration process (cf. Section 4.3 Figure 4) is completely steered by a standard Ecore generator model, the according Ecore metamodel and a set of standard JastAdd specifications. For constructing, manipulating, validating and reasoning about the input metamodel and semantic specifications all the tools available for the respective artefact can be reused. Furthermore, the process reuses the EMF and JastAdd tooling for code generation and all applied refactorings and code merges retain the metamodel implementation's API.

Consequently, the key capabilities metamodelling abstraction (**MM 1**), metamodel implementation generators (**MM 3**), semantics abstraction (**AG 1**) and semantics generators (**AG 3**) are provided by JastEMF.

JastEMF also provides metamodelling tooling compatibility (**MM 5**). This is well demonstrated in the SiPLE-Statemachines case study by:

– Using the model-driven template language XPand to generate SiPLE code for statemachines (**MM 5.2**). In general, model-driven template languages heavily benefit from semantics-integrated metamodelling, because computed semantics can be reused within templates.
– Using EMFText to generate a text-to-model parser (**MM 5.3**).
– Using the generic, tree-based EMF model editor shipped with EMF that seamlessly integrates semantics in its properties view (**MM 5.4**). Thus, models can be interactively edited, their semantics browsed and semantic values manually changed, whereas dependend semantics are automatically updated.
– Using EMFText and EuGENia/GMF to generate advanced SiPLE and SiPLE-Statemachine editors with integrated semantics (**MM 5.5**).

Regarding metamodelling and semantics consistency (**MM 2**, **AG 2**) JastEMF's integration approach has to be evaluated from two perspectives: First, the individual consistency of the Ecore metamodel and the JastAdd specifications should and can be checked reusing their respective tooling. Second, in semantics integrated metamodelling the consistency of the mapping between syntax and semantics specifications (cf. Figure 3) has to be considered. As the JastEMF integration process (cf. Figure 4 $\boxed{\mathbf{2}}$) automatically derives a JastAdd AST specification from the Ecore metamodel, JastEMF provides such consistency for concepts in E_{syn_Ω}. However, JastEMF does not yet check the correctness of the semantics mapping ($E_{sem_{\Omega}}$), which we like to improve in the future by an additional analysis step integrated in JastEMF.

Metamodel and model compatibility (**MM 4**) and semantics modularity (**AG 4**) both relate to extensibility, reuse and modularisation. In the EMF, metamodel and model compatibility helps to integrate *existing* languages, their tooling and models — i.e., to reuse *existing* metamodel implementations. Therefore, Ecore supports importing and referencing metaclasses and their implementation from other metamodels. For such reuse scenarios JastAdd has no appropriate mechanism. AST specifications can be combined from several specifications, but JastAdd always generates a new evaluator implementation and does not support the reuse of existing AST classes and their semantics. Consequently, reuse can only be achieved on the specification, but not implementation level. This limitation could be addressed by extending the JastAdd AST specification language with a packaging and package import concept that conforms to Ecore's metamodel imports[8].

On the other hand, JastAdd's aspect mechanism and weaving capabilities permit semantic extensions of languages by contributing new attributes (and node types) to an existing abstract syntax. As the EMF does not support the external contribution of structural features to existing metaclasses, the original metamodel needs to be changed to incorporate semantic extensions. This is a severe drawback considering incremental metamodel evolution that motivates further research on advancing modularity in future metamodelling approaches.

6.2 Limitations and Further Issues

JastAdd Rewrite Issues JastAdd is not only a RAG system, but also supports local, constrained rewrites that are executed as soon as a node with an applicable rewrite is accessed. Within rewrites new nodes can be constructed and existing ones rearranged. We observed that in EMF such AST rearrangements in the combination with tree copies can lead to broken ASTs. Therefore, JastEMF does not support JastAdd rewrites.

Semantics of Incomplete Models In dynamic environments such as the EMF, *syntactically* incomplete models are common throughout editor sessions. However, semantics of syntactically erroneous models are not defined and typically their evaluation fails with an exception. To our experience, most interactive modelling tools do not shield editor users before such exceptions. There is a need for more sensitive consideration of semantics in metamodelling frameworks and associated tooling, such that users are not disturbed by semantic exceptions caused by syntactically erroneous structures.

For future work, we consider the investigation of incremental AGs [23, 24], which trace attribute dependencies to reduce the recomputation overhead in the presence of frequent context information changes, to address these issues. Metamodelling tools

[8] To by-pass the problem, we introduced simple helper properties and attributes in the SiPLE-Statemachines case study. The properties hold specified entry actions, guard expressions and transition actions as ordinary Java strings whereas the attributes initialise SiPLE's parser to transform these strings into appropriate SiPLE ASTs.

could use their attribute dependency knowledge to decide whether an attribute is defined or not. If an attribute is not defined — i.e., depends on missing model parts — its evaluation could be delayed to prevent it from throwing an exception.

In summary, we think, that JastEMF's benefits clearly outweigh the remaining technical problems. It demonstrates, that RAGs contribute declarativeness, well-foundness, generativeness and ease of specification for semantics-integrated metamodelling. On the other hand, metamodels and their accompanying frameworks provide convenient means to specify the API of AG generated tools and prepare their integration into software development platforms.

7 Related Work

There are a number of approaches related to and dealing with metamodel semantics. In particular we distinguish related work that (1) can benefit from our approach, like concrete syntax mapping tools and (2) propose alternative solutions, like constraint languages, integrated metamodelling environments, graph-grammars or abstract state machines. In the following we investigate each of them.

Textual concrete syntax mapping tools like EMFText [9] and MontiCore [10] combine existing parser generator technology with metamodelling technology to realize text-to-model parser generators [25]. They enable users to generate powerful text editors including features such as code completion and pretty printing. However, semantics analysis is often neglected or involves proprietary meachanims for implementing a subset of static semantics like name analysis manually. Such tools could immediately profit from our integration of formal semantics in metamodelling.

Constraint languages like the OCL [16] or XCheck [7] enable the specification of well-formedness constraints on metamodels. The rationale behind OCL is to define invariants that check for context-sensitive well-formedness of models and to compute simple derived values. However we are not aware of any application of OCL for the specification of complete static semantics. In comparison AGs do not focus on constraint definitions, but are widely applied for semantics specification and provide advanced means to efficiently derive the context-sensitive information language constraints usually depend on[9].

Integrated metamodelling environments provide dedicated languages to specify abstract syntax and semantics but often lack a formal background. Usually semantics have to be specified using a special constraint language and a special operational (i.e. imperative) programming language, both tightly integrated with a metamodelling language and its framework.

[9] E.g., consider the specification of a data-flow analysis as presented in [26].

A typical representative is Kermeta [27]. A language in Kermeta is developed by specifying its abstract syntax with an Essential MOF (EMOF) metamodel and static semantics with OCL constraints. Execution semantics can be implemented using a third 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 cross-cutting semantic concerns. Additionally, Kermeta and JastEMF projects immediately benefit from EMF tooling in Eclipse.

Our main concern about such integrated metamodelling environments is the overhead for developers to learn and apply all their different proprietary languages. We believe that JastAdd's seamless integration with Java has two main benefits: (1) one can rely on Java's standardised and well-known semantics and (2) the smooth learning curve from Java to declarative semantics specification reduces the initial effort for using JastAdd.

Graph-grammars are a convenient approach not only to specify structures — i.e. metamodels' abstract syntax — but also operations on these structures — i.e. metamodel semantics. Given such specifications, **graph rewrite systems** can be used to derive appropriate repository implementations and semantics [28]. The main advantages of the graph-grammar approach are its well-founded theoretical background and its uniform character where syntax and semantics can be specified within a single formalism.

PROGRESS An important research project, that exploited graph-grammars for tool development and integration, has been the IPSEN project [28]. Its programming language PROGRESS (PROgramming with Graph REwriting Systems) supports the specification of graph schemas (i.e. metamodels), graph queries and graph transformations (i.e. semantics). PROGRESS graph schemas rely on attributed graph grammars to specify node attributes and their derived values. Though, attributed graph-grammars should not be confused with AGs. Attributed graph-grammars have no distinction between inherited and synthesized attributes and consequently lack many convenient AG concepts like broadcasting. In summary, the IPSEN project demonstrated, that graph-grammars are a convenient formalism to specify a broad range of tools and automatically integrate their repositories and semantics.

FUJABA A more recent graph rewriting tool is FUJABA [29], which integrates Unified Modelling Language (UML) class diagrams and graph rewriting to specify semantics of class operations. It provides *story driven modelling* as a visual language to define rewrite rules, which can be compared to UML activity diagrams. MOFLON [30] adapts FUJABA to support the Meta Object Faclility (MOF) as a modelling language.

In general we think, that graph-rewriting systems are harder to understand than AGs. Given a set of rewrite rules, it is complicated to foresee all possible consequences of their application on start graphs. Rewrite 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 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 [15] (*non-terminal* attributes in JastAdd) or JastAdd's local rewrite capabilities which improve in that direction.

Abstract State Machines (ASMs) are a theoretically backed approach to specify execution semantics [14]. For the specification of metamodel semantics they were recently applied in [31] to define sets of minimal modelling languages with well defined ASM semantics — so called *semantic units*. The semantics of an arbitrary modelling language L can now be defined by a mapping of L to such semantic units (*semantic anchoring*). Of course, the transformation to semantic units and context-sensitive well-formedness constraints (i.e., static semantics) still have to be defined using other approaches. Thus, ASMs and our RAG approach complement each other for the purpose to specify metamodels' execution *and* static semantics.

8 Conclusion

In this paper we presented the application of RAGs for metamodel semantics. We sketched necessary foundations — essentially that most metamodelling languages can be decomposed into context-free and context-sensitive language constructs — and presented JastEMF, an example integration of the EMF metamodelling framework and the JastAdd RAG system. Finally, we demonstrated and evaluated the advantages and limitations of our approach by a case study, which is exemplary for both compiler construction (SiPLE) and metamodelling (statemachines). This shows, that for MDSD the well-investigated formalism of RAGs is a valuable approach for specifying metamodel semantics and on the other hand, MDSD introduces interesting application areas and new challenges for RAG tools.

Acknowledgments

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*, the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) within the project *Hyper-Adapt*, and the European Commission within the FP7 project MOST #216691.

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

References

1. Selic, B.: The Theory and Practice of Modeling Language Design for Model-Based Software Engineering – A Personal Perspective. In: GTTSE '09. LNCS, Springer (2010) to appear.

2. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF – Eclipse Modeling Framework. 2 edn. Addison-Wesley (2008)
3. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) **24**(3) (2000) 301–317
4. Knuth, D.E.: Semantics of Context-Free Languages. Theory of Computing Systems **2**(2) (1968) 127–145
5. Ekman, T., Hedin, G.: The JastAdd system — modular extensible compiler construction. Science of Computer Programming **69**(1-3) (2007) 14–26
6. ATLAS Group: ATLAS Transformation Language (ATL) User Guide. http://wiki.eclipse.org/ATL/User_Guide (2006)
7. Efftinge, S. et al.: openArchitectureWare User Guide v.4.3.1. http://www.openarchitectureware.org/pub/documentation/4.3.1/ (2008)
8. Oldevik, J.: MOFScript User Guide v.0.8. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide-0.8.pdf (2009)
9. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: ECMDA-FA '09, Springer (2009) 114–129
10. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: TOOLS '08 (Objects, Components, Models and Patterns). Volume 11 of LNBIP., Springer (2008) 297–315
11. Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit. (2006)
12. Kolovos D.S.: Editing EMF models with Exeed (EXtended Emf EDitor). www.eclipse.org/gmt/epsilon/doc/Exeed.pdf (2007)
13. Ekman, T.: Extensible Compiler Construction. PhD thesis, University of Lund (2006)
14. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
15. Vogt, H.H., Swierstra, D., Kuiper, M.F.: Higher Order Attribute Grammars. In: PLDI '89, ACM (1989) 131–145
16. OMG: Object constraint language, version 2.2. http://www.omg.org/spec/OCL/2.2/ (2010)
17. Farrow, R.: Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-defined, Attribute Grammars. In: SIGPLAN '86, ACM (1986) 85–98
18. Boyland, J.T.: Remote attribute grammars. Journal of the ACM **52**(4) (July 2005) 627–687
19. Magnusson, E.: Object-Oriented Declarative Program Analysis. PhD thesis, University of Lund (2007)
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP '01. LNCS, Springer (2001) 327–353
21. Bürger, C., Karol, S.: Towards Attribute Grammars for Metamodel Semantics. Technical report, Technische Universität Dresden (2010) ISSN 1430-211X.
22. Hedin, G.: Generating Language Tools with JastAdd. In: GTTSE '09. LNCS, Springer (2010) to appear.
23. Reps, T., Teitelbaum, T., Demers, A.: Incremental Context-Dependent Analysis for Language-Based Editors. ACM (TOPLAS) **5**(3) (1983) 449–477
24. Maddox III, W.H.: Incremental Static Semantic Analysis. PhD thesis, University of California at Berkeley (1997)
25. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: ECMDA-FA '08. Volume 5095 of LNCS., Springer (2008)
26. Nilsson-Nyman, E., Hedin, G., Magnusson, E., Ekman, T.: Declarative Intraprocedural Flow Analysis of Java Source Code. Electron. Notes Theor. Comput. Sci. **238**(5) (2009) 155–171
27. Muller, P., Fleurey, F., Jézéquel, J.: Weaving Executability into Object-Oriented Meta-languages. In: MpDELS '05. Volume 3713 of LNCS., Springer (2005) 264–278
28. Nagl, M., ed.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Volume 1170 of LNCS. Springer (1996)

29. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE '00, ACM (2000) 742–745
30. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard Compliant Metamodeling Framework with Graph Transformations. In: ECMDA-FA '06. Volume 4066 of LNCS., Springer (2006) 361–375
31. Chen, K., Sztipanovits, J., Abdelwahed, S., Jackson, E.K.: Semantic Anchoring with Model Transformations. In: ECMDA-FA '05. Volume 3748 of LNCS., Springer (2005) 115–129