An Attribute-based Approach to the Analysis of Model Characteristics

Christian Saad, Florian Lautenbacher and Bernhard Bauer

Programming Distributed Systems Lab, University of Augsburg, Germany

Abstract. Modeling languages provide a powerful technique for describing domain-specific concepts and their relationships. Since the syntactical structure is determined by a meta-model, expressions can be defined on the meta layer and evaluated for arbitrary instances. Currently, meta-modeling standards lack an easy way for defining rules capable of dynamically examining the behavior of a model. In this paper we discuss a new approach that links meta-modeling with well-understood methods from the field of compiler construction in order to express semantic constraints and perform data-flow calculations on model instances. We show how our approach can be used to analyze and validate business processes and specify further use cases like e.g. the calculation of model metrics.

1 Introduction and Motivation

Meta-models capture the concepts of an application domain in a formal way. Widely utilized in computer science - most notably in the field of software engineering - models may describe (software) systems, interactions or control and data flows. While mainly serving as documentation in the past, practices like model-driven engineering (MDE, [1]) leverage their usage, e.g. through automatic code-generation.

Static semantics, i.e. restrictions on instances that depend on the context in which they appear and therefore cannot be expressed using solely syntactical constructs, play an important role in the modeling process. The Object Constraint Language (OCL, [2]) is a prominent method allowing to define such restrictions. It is fully integrated with other OMG technologies like the Model-Driven Architecture (MDA$^{\text{TM}}$), OMG's approach to MDE, that intends to enhance reuse and automation in software development. However, making heavy use of a language's syntactic structure, like the OCL does, has the disadvantage of being highly sensitive to changes of the modeling language. Furthermore, since OCL does not allow for evaluation of cyclic data flows, possibilities for analyzing dynamic aspects, e.g. with an abstract interpretation, are severely limited.

The approach discussed in this paper overcomes these shortcomings by combining concepts from the worlds of meta-modeling and formal languages, both of which are methods allowing to define domain specific languages (DSL) through different layers of abstraction. This enables to adapt two well-understood techniques from the field of compiler construction, attribute grammars (AG) and data-flow analysis (DFA), for the purpose of enriching (meta-)model entities with semantic attributes describing local data flows. Evaluating the attributes for an arbitrary model instance then yields information about its dynamic characteristics.

This paper is structured as follows: The relevant basics coming from the fields of compiler construction and meta-modeling are outlined in Section 2. In Section 3 we examine the common grounds between meta-models and context free grammars in order to derive the requirements for semantic attributes on meta-models before proposing an exemplary specification. The usefulness of the presented approach is discussed in the following section in the context of an implemented use case. Section 5 gives an overview on related work, before we show how our approach can be further extended.

2 Background

2.1 Attribute Grammars

Attribute grammars - introduced by [3] - specify the static semantics of a programming language L_{PL} on top of its syntactic structure defined by a context free grammar (CFG) $G_{PL} = (N, T, S, P)$, e.g. using the Extended Backus-Naur Form (EBNF). Its productions P describe how non-terminals N, starting with $S \in N$, can be replaced by other (non-)terminals, until a string is left that consists only of terminal characters T.

An attribute grammar AG adds to a CFG a set of attributes A, each of which is assigned to a symbol $x \in N \cup T$ and is either of the type Inh (inherited) or Syn (synthesized). Semantic rules R assigned to productions P calculate an attribute's value by taking other attributes in the same production as input arguments and mapping them to a result value by applying a function f. This can be written as $X_i.a = f(X_j.b, ...)$.

The abstract syntax tree (AST) of a program $prog \in L_{PL}$ has non-terminals N as nodes and terminals T as leafs. The attribute-enriched version is called decorated AST (DAST). Attribute instances can be thought of as property fields of the nodes and leafs while the semantic rules establish input dependencies between attributes at adjacent nodes, thereby describing an information flow along the AST's structure. To obtain an unambiguous result, the derived equation system must not contain cycles. Usually, this is achieved through the use of restricted classes of AGs that allow only non-cyclic dependencies and have a (optimal) static evaluation order that can be computed in advance (cf. [4]).

2.2 Data-flow Analysis

Compilers use data-flow analysis typically for optimization and error detection purposes, e.g. for eliminating unreachable code or for releasing memory assigned to variables after their last access. DFA allows for an abstract interpretation, because it is able to approximate some aspects of the runtime behavior of a program.

Data-flow equations (specific to the DFA problem to be solved) assigned to nodes $n \in N$ of a program's control flow graph $G = (N, E)$ are able to access results calculated at adjacent nodes and use them as input arguments. DFA equations usually operate on value sets that are initialized with a default value (typically $\emptyset$ or the value domain D) and merged using set operations ($\cup$ or $\cap$), both of which are specific to the current evaluation semantics. Cyclic definitions are solved by iteratively recalculating the equation system and propagating the results after each iteration to neighboring nodes until the resulting values stabilize in an algorithm-dependent fixpoint (cf. [5]).

As an example, the equations $in(n) = \bigcap_{x \in Predecessors(n)} out(x)$ and $out(n) = gen(n) \cup (in(n) - kill(n))$ applied to nodes N propagate information that is locally generated ($gen(n)$) but not destroyed ($kill(n)$) along the flow direction of the graph. $in(n)$ aggregates the result sets of all predecessors using the operator $\cap$, narrowing down the final fixpoint set to values that reach a node on *all* of its incoming edges.

2.3 Meta-models

As the Meta-Object Facility (MOF, [6]) specification states: *"A meta-model is a model used to model modeling itself"*. Meta-models like the MOF-based Unified Modeling Language (UML) define an - often extensible - modeling language that incorporates the syntax and part of the corresponding semantics for a specific application domain.

Each element in a model is an instance of its meta-model class. A meta-modeling architecture may comprise an arbitrary number of meta layers, since each layer may have an overlaying specification. For practical purposes MOF is limited to four layers M0-M3 with the meta-meta layer M3 consisting of the self-describing MOF model.

Since many DSLs require restrictions that cannot be expressed through the syntactical constructs of the meta language, meta-models are often complemented with static semantics definitions that can be described informally but also through constraint languages like the OCL.

3 Semantic Attributes for Meta-models

3.1 Attributes in the Context of Modeling

Grammars and models share some basic principles: Just as the words that make up a CFG's language follow the structure determined by the grammar's productions, meta-models characterize modeling languages in terms of how elements in the domain are related to each other. Identifying similarities and differences between both worlds forms the basis for devising an attribution concept for meta-models.

First, there has to be an agreement on the alignment of the abstraction layers. The authors of [7] describe a bridge between model- and grammarware, linking the two self-describing "meta-languages" EBNF and MOF M3 and connecting grammars and programs to the M2 (meta) and M1 (model) layers respectively.

Figure 1 shows where attributes fit in: The basic idea is to assign attributes to meta-model elements the same way attributes are connected to grammatical productions and to evaluate them for a given AST or model. Probably the most obvious requirement here is the need for an attribution language, the term "language" already hinting at the fact that it must be located at a meta-layer (M3 in this case), meaning we have to provide syntax (abstract and concrete) and semantics for the definition of attributes.

While there is no official standard for the syntax of attribute grammars, some authors like [8] propose an extended definition of CFGs that incorporates declaration and usage directives. The same can be applied to meta-models, resulting in the enrichment of the MOF definition with "meta-attributes". Since this is likely to cause problems with existing tools and algorithms, e.g. model transformations, it is advisable to keep

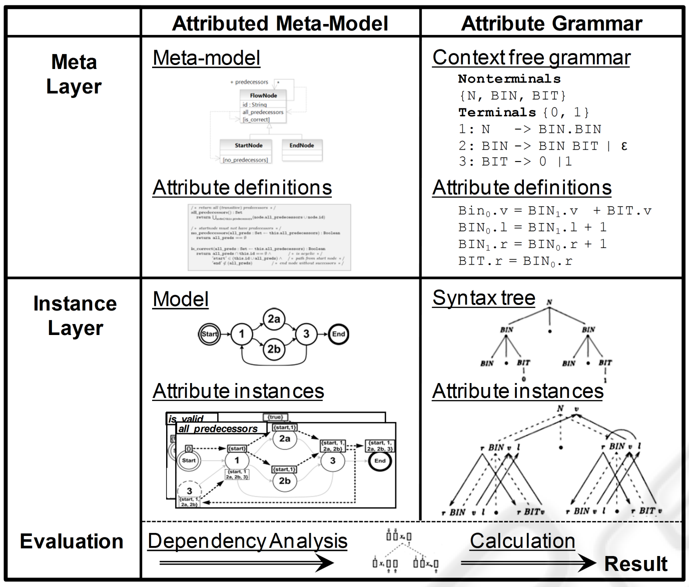


Fig. 1. Comparing attributed meta-models and attribute grammars on a conceptual level (the attributed (meta-)model is shown in detail in Figure 3, the AG example can be found in [8]).

attributes (and attribute definitions) outside of the (meta-)models and instead using a framework to establish the links between model elements and their attributes.

In grammars, attributes exist as (data) type definitions assigned to symbols $x \in N \cup T$ creating an occurrence every time a symbol is used in a production. Occurrences yield attribute instances in the AST each time the production is applied. With models, an attribute definition, consisting e.g. of name and data type, is referred to by attribute occurrences attached to model objects. These occurrences have to be declared manually since there is no equivalence for (non-)terminal definitions in meta-models.

A key difference between AGs and meta-models lies in their internal structure: A CFG correlates to a push-down automaton and its language expressions to ASTs while models and meta-models are usually graphs consisting of nodes and edges. As a direct impact, the distinction between synthesized and inherited attributes made in AGs is not feasible for models, since a model graph has no distinguished up or down direction. The information flow is therefore solely formed by the input dependencies of the semantic rules which allows for a feature not found in traditional AGs: Semantic rules can access attributes located at arbitrary model objects using more sophisticated input selectors, e.g. an all operator collecting attributes from all instances of a class (cf. OCL).

These selectors for required input arguments play a more important role in attributing meta-models than in AGs. Since only attributes assigned to symbols of the same production can be used as input and both sides of a grammatical production are always directly connected in the AST, there is no need for a complex selection mechanism. For (meta-)models on the other hand, it is not immediately clear how a corresponding "neighbor" relationship between attributes may look like. In fact, as mentioned above for the all selector, attribute instances may be referenced from everywhere in the model.

Keeping in mind that associations denominate important relationships in the application domain, a method for specifying input arguments should favor routing attributes along these connections but also have the possibility of addressing arbitrary attributes if necessary. As further complication, due to the multiplicity property of associations multiple attribute instances (or none at all) may be returned. Because of its extensive set of navigation operations, a simplified OCL - optimized for addressing neighboring objects and extended with the ability to access and filter their semantic attributes - presents itself as a viable alternative to the implementation of a proprietary selection language.

In addition to the arguments, attribute occurrences must be connected to the semantic rules that are going to process the instance values as provided by the selector during the evaluation process. The rules can be invoked externally through the framework or expressed using a platform independent language. Adding support for reflective examination of the (meta-)model and the attribution provides additional flexibility.

To support generalization, we have to agree on how attributes are to be handled in an inheritance hierarchy. Keeping in line with the common modeling notion where subclasses inherit properties from superclasses, we're adding semantic attributes to that set. A more sophisticated solution may support overriding of attribute occurrences much like functions in object-oriented programming languages can be overridden. Introducing this polymorphism affects the instantiation mechanism, since based on the concrete realization type of a model object, different sets of semantic rules have to be invoked.

A concrete syntax acts as interface to a language and should therefore be also designed carefully. Although a textual representation would be possible (and is necessary for internal processing), visual illustration tends to be much more intuitive, especially in the area of meta-modeling where working with graphical tools is state-of-the-art. The concrete syntax of the language constructs should therefore fit in with the representation of (UML) model objects showing attribute occurrences along with their definitions, their connection with semantic rules and (optionally) their input dependencies.

Once the attribution has been instantiated for a given model (resulting in a decorated model) the semantic rules have to be invoked with respect to the defined input dependencies. For non-circular definitions this can be done in a single run while cyclic dependencies require multiple passes until all values have stabilized in the desired fixpoint. A simple algorithm for this problem is introduced in the next section.

3.2 Defining an Attribution Concept for Meta-models

In this section we provide a definition for attribute enriched meta-models which may serve as basis for a more extensive formalization. Because of its relevance and wide - spread use, a MOF-based layout was chosen to internally represent the enhanced meta-model.

Figure 2 shows a simplified view on MOF M3, extended with the definition of semantic attributes: A MOF *Classifier* is linked with *AttributeOccurrence*s whose properties are controlled by an *AttributeDefinition*. The instance value of an *AttributeOccurrence* is calculated by a semantic rule of the type *Assignment* while *SemanticConstraint* rules just evaluate to true or false. A reference to the implementation of a *SemanticRule* is given in its *id* property. The input dependencies are part of the abstract syntax and modeled through the *uses* association and a simple *Selector* mechanism allowing

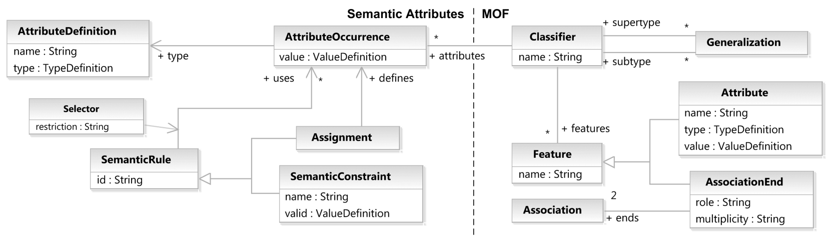
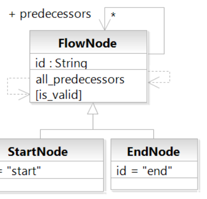
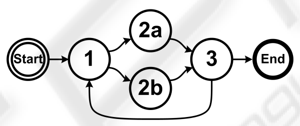


Fig. 2. Simplified representation of the MOF extended with the definition of semantic attributes.

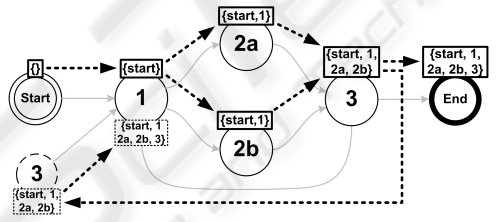
for further restrictions. Our selector directives enable for example to narrow down the set of returned attribute instances to instances located at neighbor objects (i.e. objects connected through an association), instances at supertypes or attributes at the local instance. Since the additional elements are only required for computation purposes the extensions can be applied dynamically before the evaluation (as shown in Section 4) thereby staying completely MOF-compliant from the perspective of the source models.



(a) Meta-model of a flow-graph supporting alternative paths



(b) A flow-graph model violating the demand for sequential flows



(c) Dependency chain for the attribute $all\_predecessors$

Fig. 3. Example for attribution, instantiation and evaluation of a flow-graph (meta-)model.

Figure 3(a) shows a meta-model describing a simple flow-graph in the chosen concrete syntax: Semantic attributes and constraints are displayed in an additional compartment using the name of their attribute definition or constraint name while input dependencies are visualized through dashed arrows. In this example we want to allow sequential alternative flows, i.e. there must not be any back-edges. This is ensured with use of the semantic attribute $all\_predecessors$ that carries the transitive list of all predecessors. If the constraint is_valid finds that the current node is already contained in this set, a circular flow is detected. This reveals not only the fact that the requirements have been violated but also the exact position where the problem is located. Using a mathematical notation, the semantic rules can be written as follows:

```
/* compute all (transitive) predecessors of a flownode */
all_predecessors: Assignment on FlowNode
   return ⋃_{node∈this.predecessors}(node.all_predecessors ∪ node.id)

/* check validity of flownode */
is_valid: SemanticConstraint on FlowNode
   declare this.all_predecessors as all_preds : Set
   return all_preds ∩ this.id == ∅ ∧               /* does not contain cycle */
          'start' ∈ (this.id ∪ all_preds) ∧         /* require path from start node */
          (this.id ==' start' → all_preds == ∅) ∧  /* start node without predecessors */
          'end' ∉ (all_preds)                      /* end node without successors */
```

Here, the input dependencies corresponding to the *uses* and *Selector* declarations (cf. Figure 2) are stated implicitly through access on attribute occurrences. Since canonical MOF attributes do not change during an evaluation they can be treated as constants.

Computing the result values for attributes requires the following tasks: First the attributes have to be instantiated for the given model. Based on the dependency relationships which are specified in a declarative way, a valid execution order for the application of semantic rules has to be determined before executing them in this sequence, storing result values and passing them as input arguments to other semantic rules.

The instantiation of the input dependencies has to be carried out with respect to the selector statements. The execution order can be established by performing a depth-first search on the dependencies for a given attribute instance. Omitting cyclic references (back edges) returns a directed acyclic graph (DAG) with a single root element - referred to as dependency chain - that can be evaluated in a single bottom-up run, allowing to reuse computed results in other "branches" of the DAG. Cyclic definitions require an iterative evaluation, where after a run the results are propagated along the hitherto ignored back-edges and the evaluation is repeated until a fixpoint has reached.

Please note that a model may contain multiple dependency chains and also that a dependency chain may be split up into several subchains which can happen if the root attribute instance is not the absolute root of the whole chain (e.g. if the root element has been chosen randomly). However, this has no effect on the final result because the backtracking of the input dependencies ensures that all strongly connected components (SCC) are completely contained in the examined part of the chain. Therefore, the evaluation of each chain can be considered to be an atomic process.

In the average this algorithm can be expected to perform much better than iterating over the whole equation system repeatedly and invoking semantic rules with known input values (data driven approach).

Figure 3(c) shows the dependency chain of *all_predecessors* for the flow-graph 3(b). The cyclic dependency is represented through a virtual copy of node 3 offering the first-pass result from node 3 as input for node 1 in the second run, where the constraint *is_valid* then detects an error due to "1" already being in the list of predecessors.

4 Implementation and Application Areas

In this section we describe our prototypical evaluator in the context of a business process oriented use case and discuss other areas where semantic attributes can be applied.

To fulfill the use case's requirements, actions that are definitely executed during an arbitrary run of a workflow should be detected. This corresponds to the minimal re-

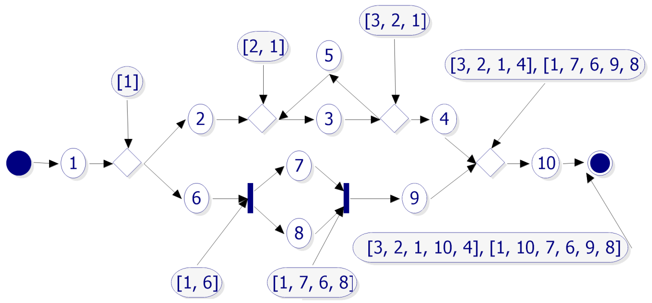


Fig. 4. The calculated sets of minimally executed actions for this workflow.

sources that have to be invested in a workflow run and reveals optional steps, helping to improve the arrangement of the business actions. The Eclipse Java Workflow Tooling (JWT, [9]) framework was chosen as workflow designer, since it is based on the Eclipse Modeling Framework (EMF, [10]) - a (partial) implementation of the MOF specification - and can be easily extended, e.g. through an aspects-mechanism that allows to alter the underlying meta-model. Finding actions on minimal paths is far from trivial, since parallel and alternative paths - possibly even nested - may narrow down this set, introduce alternative sets or create "shortcuts". DFA allows for a pretty straightforward fixpoint-based solution, merging action sets at parallel end nodes and creating new alternatives by combining incoming results using the cartesian product. An example of this evaluation is shown in Figure 4. The annotation at the final node indicates that two distinct minimal execution paths exist, both of which do not contain action 5.

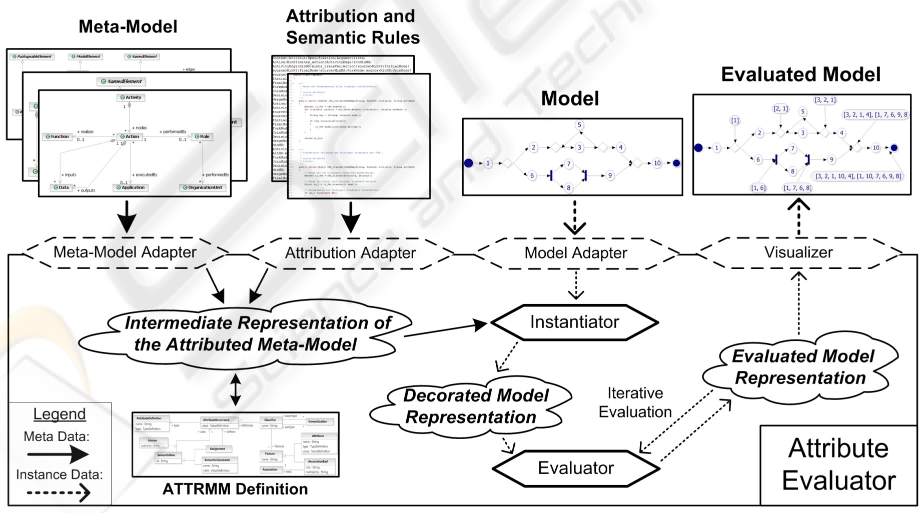


Fig. 5. Architecture of the implemented EMF-based attribute evaluator prototype.

The architecture of the prototype that runs this use case is shown in Figure 5. The external components - meta-model, model, attribution - are transformed into an internal

EMF representation based on the previously introduced abstract syntax. Currently supported attributions consist of a textual specification and a Java implementation of the semantic rules. An instantiator module creates the decorated model and passes it to the evaluator that implements the presented iterative algorithm.

We are currently working on other use cases including a modified DFA able to calculate all *possible* execution paths. Complemented with validations ensuring constraints like the correct nesting of parallel paths, the results serve as a powerful source for simulation and error detection purposes, e.g. identifying worst-case resource requirements.

Other application areas include block identification in graphs through SCCs, detecting unneeded fragments similar to dead code elimination in compilers and calculating model metrics [11] or the compliance with a set of modeling guidelines [12].

5 Related Work

There are already attempts to use attribute grammars in model-driven engineering: [13] shows that attribute grammars can be used for model transformations. The authors lift the source model to a textual form of the abstract syntax and transform it using a so-called evaluator into another textual abstract syntax describing the target model. Another example presented in [14] aims to improve the measurement of process modeling using attribute grammars to specify the measurement metrics. [7] and [15] align the CFG definition with MOF M3 in order to automatically generate mappings between grammars and meta-models. The problem of dynamic characteristics is also addressed by [16], who use automatically generated instances of meta-models as basis for tests.

None of these approaches extends meta-models with attributes in order to use them for DFA or other purposes as we propose in this paper although DFA-based approaches have been discovered to be useful for models (cf. [17]).

Additional work not connected to MDE but nevertheless relevant in the context of this approach can be found in the area of attribute grammar research, including circular AGs that support DFA [18], supported for example by the JastAdd framework [19].

6 Conclusions and Future Investigations

In this paper we have presented an approach that introduces semantic attributes into modeling as a formal method to perform a static analysis of a model's dynamic behavior. Since it relies on passing on information locally, complex navigation statements can be avoided while analysis of (dynamic) circular information flows becomes possible.

We discussed how semantic attributes can be transferred to the world of models, outlining the challenges encountered and suggesting possible solutions. A MOF-based declarative specification realizing the identified problems was given along with an evaluation algorithm for circular dependencies that can be expected to outperform a purely data driven method. The feasibility was demonstrated in the context of several use cases, in particular the abstract interpretation of workflows which is already supported by a prototype serving as proof-of-concept and as basis for future research work.

As this paper is intended to provide an overview of the requirements and challenges connected to the usage of semantic attributes for the evaluation of meta-models, there exist a variety of starting points for enhancements: It is planned to extend the prototype with advanced support for generalization and data types and to include the JastAdd framework for attribute evaluation. Whether a static evaluation order can be derived from an attributed meta-model still has to be researched. A formal language definition is required as well as a method for expressing attribute selection statements, preferably as an extension of OCL.

The approach described in this paper constitutes a suitable foundation for all these enhancements. To provide input for the future development, further use cases in the area of MDE will be identified and evaluated.

References

1. Kent, S.: Model Driven Engineering. In: Proceedings of the Third International Conference Integrated Formal Methods (IFM'2002). (2002)
2. Object Management Group: Object Constraint Language. Specification Version 2.0 (Mai 2006)
3. Knuth, D.E.: Semantics of Context-Free Languages. Theory of Computing Systems, 2 (2) (June 1968) 127–145
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1986)
5. Allen, F.E.: Control flow analysis. SIGPLAN Not., 5 (7) (1970) 1–19
6. Object Management Group: Meta-Object Facility. Specification Version 2.0 (January 2006)
7. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 159–168
8. Wilhelm, R., Maurer, D.: Compiler Design. Addison-Wesley (1995) Second Printing.
9. The Eclipse Foundation: Eclipse Java Workflow Tooling (JWT), http://www.eclipse.org/jwt/
10. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson (2003)
11. Chidamber, S.R., Kemerer, C.F.: Towards a Metrics Suite for Object Oriented Design. SIG-PLAN Not., 26 (11) (1991) 197–211
12. Ambler, S.W.: The Elements of UML 2.0 Style. Cambridge University Press (2005)
13. Dehayni, M., Féraud, L.: An Approach of Model Transformation Based on Attribute Grammars. In: 9th International Conference on Object Oriented Information Systems (OOIS). Volume 2817 of Lecture Notes in Computer Science. (2003) 412–423
14. Atan, R., Ghani, A.A.A., Selamat, M.H., Mahmod, R.: Software Process Modelling using Attribute Grammar. International Journal of Computer Science and Network Security (IJC-SNS), 7(8) (August 2007) 273–281
15. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
16. Gogolla, M., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Proceedings of the 6th Int. Conf. Unified Modeling Language, Springer (2003) 265–279
17. Garousi, V., Bri, L., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams. (2005)
18. Magnusson, E., Hedin, G.: Circular Reference Attributed Grammars - Their Evaluation and Applications. ENTCS, 82 (3) (2003)
19. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative Intraprocedural Flow Analysis of Java Source Code. In: Proceedings of LDTA 2008. (2008)