INCORPORATING SEMANTIC ALGEBRA IN THE MDA FRAMEWORK

Paulo E. S. Barbosa, Franklin Ramalho, Jorge C. A. de Figueiredo and Antonio D. dos S. Junior
Universidade Federal de Campina Grande, Departamento de Sistemas e Computação
Campina Grande, Brazil

Keywords: Metamodels, denotational semantics, formal methods and MDA.

Abstract: Denotational semantics is commonly used to precisely define the meaning of a programming language. This meaning is given by functions that map syntactic elements to mathematically well defined sets called semantic algebra. Models in semantic algebra need to be processed through reductions towards a normal-form in order to allow the verification of semantics properties. MDA is a current trend that shifts the focus and effort from implementation to models, metamodels and transformations during the development process. In order to put forward denotational semantics in the MDA vision, we turn semantic algebra into an useful domain-specific language. In this context, this paper describes our proposed MOF metamodel and ATL reductions between the generated models. The metamodel serves as abstract syntax for semantic algebra. It is useful for static semantics verifications. The reductions enable processing towards a normal-form to compare semantics. This process can be guided by using some rewrite system.

1 INTRODUCTION

Metamodels play an important role in the Model-Driven Architecture (MDA) (OMG, 2007) providing a way to unambiguously define languages. Each Domain Specific Language (DSL) can be associated to a different metamodel. A metamodel defines the constructs of the DSL in the modeling process. Thus, a model is an instance of these constructs. Metamodels are also used to define model transformation, a technique to transform one source model into a target model. Transformations play a key role in the MDA approach, allowing the transformation of the constructed model into something executable.

The formal semantics of a programming or modeling language is the assignment of meanings to the sentences or components given by a mathematical model that describes every possible computation in the language. Denotational semantics is an useful approach for precisely defining the meaning of a programming language (Schmidt, 1986). It is a mapping between a program and its meaning, called denotation. The denotation is a mathematical value, such as a number or a function. This mapping of the denotational style is feasible only if the formal notation is algebraic. The most used algebraic notation is named semantic algebra. A semantic algebra is a formalism for introducing domains and operations, being considered an abstract data type.

Simplifying or evaluating semantic models means reducing its expressions until no more reduction rules can be applied. A reduction rule receives as input a semantic model and returns as output another semantic model with some different feature. These reductions need to be applied in order to compare it with the meaning of other programs or formalisms.

The MDA architecture does not provide formal infrastructure and primitives for the definition of artifacts able to represent the required semantic correctness (Kleppe and Warmer, 2003). It deals only with the definition and manipulation of structural elements, related to the syntax defined by metamodels. It is expected that transformations involving programming and modeling languages as input and output models should have a conformance relation between models. The lack of formalization makes the production of tools difficult because semantics carries the meaning that is essential to enable automation.

In this paper we propose introducing semantic algebra in the MDA infrastructure. In order to achieve this goal, a metamodel for semantic algebra and implementation of reductions between the generated models are suggested. Both are important in the MDA vision because they: (i) give an abstract syntax to

E. S. Barbosa P., Ramalho F., C. A. de Figueiredo J. and D. dos S. Junior A. (2008).
INCORPORATING SEMANTIC ALGEBRA IN THE MDA FRAMEWORK.
In *Proceedings of the Third International Conference on Software and Data Technologies*, pages 330-336
DOI: 10.5220/0001899703300336

apply semantic algebra as a DSL; (ii) provide interoperability between semantic specifications of many languages, enabling processing of the denotations; (iii) enable formal refinement of the involved models, which can be applied both in endogenous and exogenous transformations (Czarnecki and Helsen, 2003); and (iv) are important pieces in an ongoing framework for semantic preserving in MDA transformations having as basis the well-founded theory of Denotational Semantics.

The paper is structured as follows. Section 2 presents the semantic concepts necessary to understand the proposed work. Section 3 describes the semantic algebra metamodel. Section 4 analyzes the implementation of reductions between models in this formalism. Section 5 gives an example of the metamodel instantiation. Section 6 discusses some similar works. Section 7 concludes the paper.

2 DENOTATIONAL SEMANTICS AND SEMANTIC ALGEBRA

The framework of denotational semantics, provided by Scott and Strachey (Scott and Strachey, 1971), provides a proper mathematical foundation for reasoning about programs and programming languages. It is based on denotations, that are typically a function with arguments that represent expressions before and after execution. They are defined inductively, using λ-Calculus (Böhm, 1975) to specify how the denotations of components are combined.

To give the semantics of a language, a collection of meanings is necessary. The most used framework is *Domain Theory*, which employs structured sets, called domains, and their operations as data structures for semantics. Domains plus the operations constitute a semantic algebra. According to Schmidt (Schmidt, 1986), the semantic algebra:

- clearly states the structure of a domain and how elements are used by the functions.
- modularizes and provides reuse of semantic definitions.
- makes analysis easier.

As an example, we chose a subset of the specification of an imperative programming language extracted from (Schmidt, 1986). It will also be discussed and instantiated in Section 5. This semantic algebra has four domains: (I) the *Truth Values* domain, corresponding to the boolean type and has *true* and *false* as atomic values and *not* as operation; (II) the known *Natural Numbers* domain, their uncountable elements and the basic operations *plus* and *equals*; (III) the *Identifiers* domain, commonly employed as specific memory addresses, being a set with no operations; and (IV) the *Store* domain, that models a computer store as a mapping from identifiers of the languages to their values. The operations over the store include: (i) *newstore* for creating a new store; (ii) *access* for accessing a store, and (iii) *update* for placing a new value into a store. This is the modeling of the classical concept of memory manipulation, always present in imperative languages.

I. Truth Values
  Domain $t \in Tr = \mathbf{B}$
  Operations
   true, false: Tr
   not: Tr → Tr
II. Natural Numbers
  Domain $n \in Nat = \mathbf{N}$
  Operations
   zero, one,...:Nat
   plus: Nat × Nat → Nat
   equals: Nat × Nat → Tr
III. Identifiers
  Domain $i \in Id$ = Identifier
IV. Store
  Domain $s \in Store = Id \rightarrow Nat$
  Operations
   newstore: Store
   newstore = λi.zero
   access: Id → Store → Nat
   access = λiλs.s(i)
   update: Id → Nat → Store → Store
   update = λiλnλs.[i↦n]s

3 THE SEMANTIC ALGEBRA METAMODEL

We propose a metamodel as the abstract syntax for the definition of a semantic algebra. Due to space restrictions, aspects of the metamodel were summarized. The diagram in Fig. 1 shows six MOF packages covering concepts and relationships in this domain.

The *core* package represents the identification of a semantic algebra, and how it is composed by several domains. The *metaDomains* package represents existing sets or structures which are used as basis for defining the semantic algebras. The *semanticDomains* package represents a set of elements sharing common properties. Domains may be nothing more than sets, lattices or topologies. Accompanying a domain we have a set of operations that can be considered functions. The *lambdaCalculus* package represents the λ-*Calculus* computational formalism which is responsible for function specifications. The *enrichedLambdaCalculus* package represents the inclusion of extra constructs, such as a superset, originating the *enriched lambda calculus*. It is provided through the *merge operator* because any expression in the λ-Calculus is also an expression in the enriched λ-Calculus syntax. The *abstractions* package contains specific abstractions identified in several parts of the metamodel.

The *core* package is responsible for capturing the beginning of the denotational specification. The semantic algebra concept is represented by the meta-

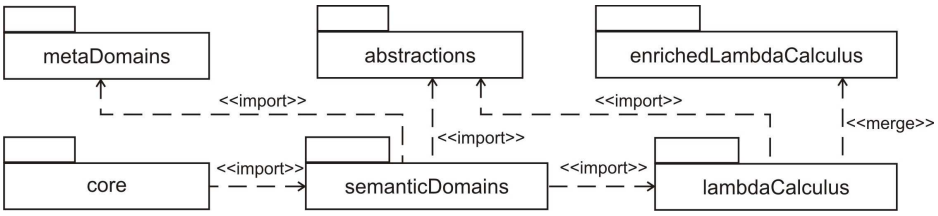

Figure 1: Semantic Algebra metamodel.

class *SemanticAlgebra* composed by zero or many *Domains* as represented in Fig. 2. This metaclass belongs to the *semanticDomains* package and is described in the next subsection.

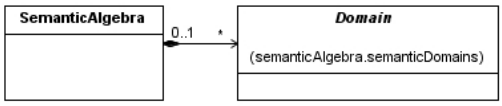

Figure 2: The core package.

These domains (semantic domains) are sets that are used as value spaces in programming language semantics. They are presented in Fig. 3. They belong to the *semanticDomains* package where are described with an underlying mathematical set (e.g. natural numbers, truth values or any other domain) which provides the isomorphism to an existing well-defined theory. This information is stored in the *Description* metaclass and in its relationship to a *MetaDomain*.

A semantic domain should be represented as a set and they can be classified according to its composition as:

- Primitive Domains: represented by the *PrimitiveDomain* metaclass. Its elements are atomic and are used as answers or semantic outputs. This kind of domain is specialized as *CharacterStrings*, *TruthValues*, *NaturalNumbers*, *Identifiers* and *Unit* only. It is incomplete because according to the described language and its paradigm, additional domains should be necessary.
- Compound Domains: represented by the *CompoundDomain* metaclass. It enables the creation of new domains from existing ones, using some domain building constructions in the Domain Theory. There are a number of assembling and disassembling operations of the compound domain. The previously defined domains are products, disjoint unions, functions and lift, a type of empty domain.
- Recursive Domains: represented by the *RecursiveDomain* metaclass. It occurs when certain programming languages features require domains whose structure is defined in terms of themselves. A recursive domain is composed by, at least, another domain that can be considered the base case of the recursion.

Since λ-Calculus is a language with a well-defined EBNF grammar, we mapped its rules to the *lambdaCalculus* package represented in Fig. 4. It has four kinds of expressions:

- Variables: represent the storage of some concept to be manipulated in the program.
- Constants: represent an extension of a suitable collection of built-in functions, including arithmetic functions, constants, logical functions, character constants and more.
- Applications: represent the denotation: the function **f** applied to the argument **x**. It is used as **f x**. In the metamodel those expressions are represented by the roles operator and operand. This format is known as *currying* and allows us to think of all functions as having a single argument.
- Lambda Abstractions: represent constructs to denote new functions. A λ-abstraction is a particular sort of expression which denotes a function. It is composed by a variable followed by the body of the function.

These expressions are represented as metaclasses and grammar constraints are, for instance, relationships between metaclasses. It is enough to represent compositionality demanded by recursive definitions.

In order to turn the operation specification into a more suitable task, the syntax of the λ-Calculus is improved with operators closer to the functional constructs. These enriched constructors are metamodeled in the *enrichedLambdaCalculus* package in Fig. 5. It is a superset of the λ-Calculus so that any expression in the λ-Calculus is also an expression in it.

In this case, *Constant* and *Variable* are now generalized into *Pattern* because of some inserted operations that treat both similarly. In addition, some extra constructs are provided, such as *let*, *case* and the considered infix operator *fat bar*. Now, a *LambdaAbstraction* receives as parameter a pattern, allowing the pattern matching concept, always present in functional languages. The *Let* metaclass is composed by

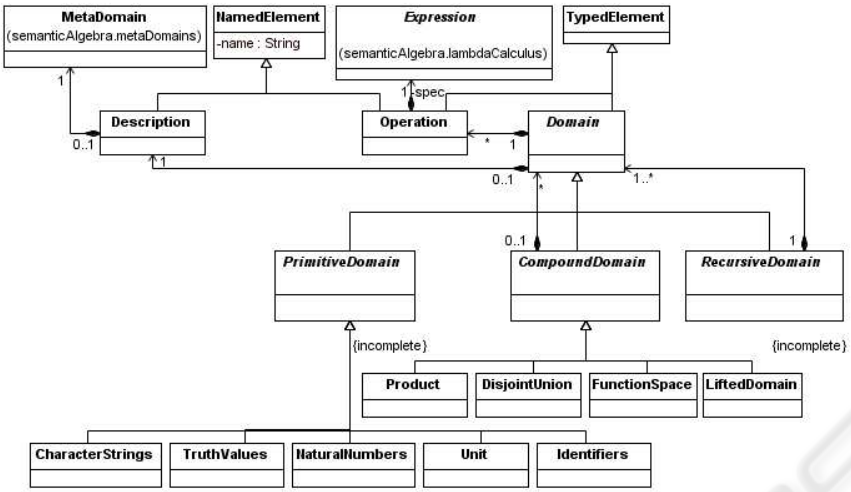

Figure 3: The semanticDomains package.

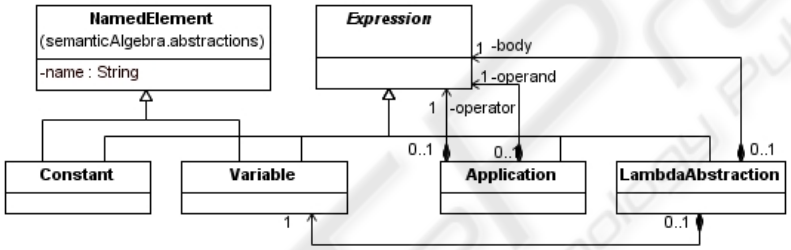

Figure 4: The lambdaCalculus package.

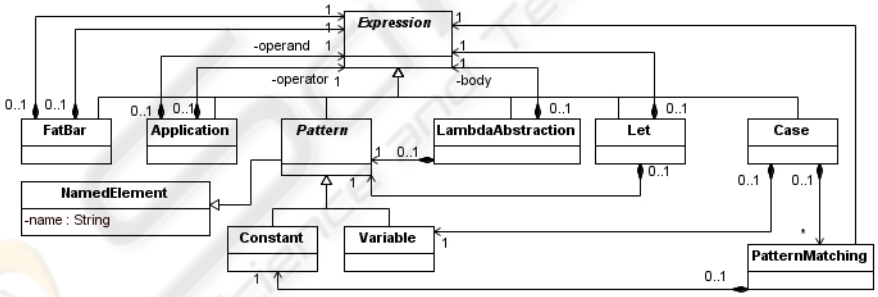

Figure 5: The enrichedLambdaCalculus package.

a *Pattern* instance that is currently instantiated and an *Expression* in which this instance will be employed. The case expression consists of a variable in which the test will happens and several *PatternMatching* instances.

4 LAMBDA REDUCTIONS

Evaluating a λ-expression is called reduction. Reductions are useful for the processing of models towards a normal-form in order to compare semantics. As λ-Calculus is inserted in the semantic algebra format, we followed the same definition. The basic idea involves substituting expressions for free variables in a similar way that parameters are passed as arguments in a function call.

In the MDA framework, we employ these reductions as refinement rules of the semantic models (instances of the semantic algebra metamodel) associated with each input and output model. These rules are applied until a normal-form is reached for each model, on which semantic properties will be verified. Therefore, these reductions will promote verify-

ing if the MDA transformations are in fact semantic-preserving. We have chosen to introduce reductions in MDA by using *model transformations*. In particular, for each reduction, we have specified an ATL module containing minor reductions specified as ATL rules. In essence, they are rewrite rules that match input models patterns, expressed as semantic algebra elements on the input metamodel, and produce output as semantic algebra patterns.

Our set of ATL transformation rules generates intermediate and final representations of semantic algebra specifications. We have implemented the main λ-reductions described in (Schmidt, 1986): α, β and η. The reductions were implemented as *ATL rules*, using *ATL helpers* for auxiliary functions or attributes. Due to the lack of space, we illustrate the β-reduction. It is useful for simplifying an expression, defining the idea of evaluation of a function application (**f x**).

In the module BetaReduction, the formula in which the β-transformation will be carried out is *(λv.E) e*. The input expression needs to be an application with the operator as an abstraction, as indicated at lines 4-5 by the input match. This is necessary to satisfy the β equation: (λv. E) E1 ⇒β E[v → E1]. The rule (*Abs2AbsFree*) at lines 8-10 deals with the input as a *LambdaAbstraction*. It invokes an auxiliary function (line 10) to discover the free occurrences of the formal parameter in the body of *f* which are to be replaced for. The auxiliary function is implemented with the ATL helper *isFree* at lines 11-15, which declares that the parameter variable is free, considering the body of the function as an expression, if: (i) it is a variable with the same name of the parameter variable; or (ii) it is an application and recursively the variable is free for the operator or the operand, or (iii) it is an abstraction with the same parameter and has a free body.

```
1:  module BetaReduction;
2:  create OUT : LC from IN : LC;
...
3:  rule Expression2Expression {
4:   from
5:    in:LC!Expression(in.isApp() and in.operator.isAbs())
6:   to
7:    out:LC!Expression
...
8:  rule Abs2AbsFree {
9:   from
10:   in:LC!LambdaAbstraction(not(self.isFree(self.v,in)))
...
11:  helper def: isFree (v:LC!Variable, e:LC!Expression):Boolean =
12:   (e.isVar() and (v.name = e.name)) or
13:   (e.isApp() and self.isFree(v,e.operator) or
14:   self.isFree(v,e.operand)) or (e.isAbs() and
15:   (not(v.name = e.parameter.name) and self.isFree(v,e.body)));
```

In order to build and validate our approach, we have adopted the ATL-DT (AI0, 2004) framework. It is an Eclipse plugin that allows in an integrated way: (i) specifying and validating metamodels; (ii) creating and validating models in conformance with their provided metamodels; (iii) specifying and executing model transformations based on metamodels and applied to valid models. These transformations are already being applied to several complex semantic models. By adopting a framework that fully supports most of the OMG standards, we promote the integration of our work with other existing tools, such as model or code editors.

5 AN INSTANTIATION EXAMPLE

In order to validate the proposed metamodel, we reused several classical specifications from (Schmidt, 1986). In Fig. 6 we illustrate the specification for imperative languages given in Section 2. The introduced labeled boxes represent instantiations of the metaclasses and the dotted arrows represent instantiations outgoing from the box and ingoing the metaclasses. First of all, one instance of the metaclass *SemanticAlgebra* is shown in the box 1 and four instances of the associated metaclass *Domain* in the box 2, corresponding to the four existing domains.

In Fig. 7 we give a general view of the instantiated elements. We deal with three existing primitive domains: *TruthValues*, *NaturalNumbers* and *Identifiers* instantiated in box 3. We also have one compound domain: *Store*, in box 4. Its operations are shown in the top of box 5, except for *Identifiers* which have no operation. It was also necessary to add to the hierarchy of the *CompoundDomain* a new subclass to represent *Store*. It was instantiated as a map from *Identifier* to a *NaturalNumber* (the two domains that together compose the store), implementing a *FunctionSpace*. A domain is also composed by a *Description* as shown in box 6. It applies the set of Natural numbers from mathematics into the *NaturalNumbers* domain, the commonly used Boolean values from programming languages into *TruthValues* and a set of identifiers composed by any character string into the *Identifiers* domain.

The remaining expressions proposed in the Store domain operations are instantiated as shown in Fig. 8. In box 7 we have two nested λ-expressions delimited by the λ symbol until the end. The first lambda abstraction receives the variable *i* as parameter and has the second lambda abstraction as expression. The second lambda abstraction receives the variable *s* as parameter and makes the function application of *s* onto *i* as its expression. The expression in box 8 follows the same idea, but the expression of a lambda abstraction is a constant, *zero* in this case. Box 9 presents the creation of an anonymous lambda abstraction, with the variable *i* as parameter, the variable *n* as expression and the anonymous function is applied to the variable

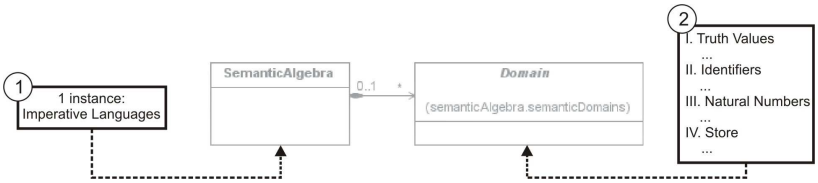

Figure 6: Instantiating the semantic algebra for imperative languages.

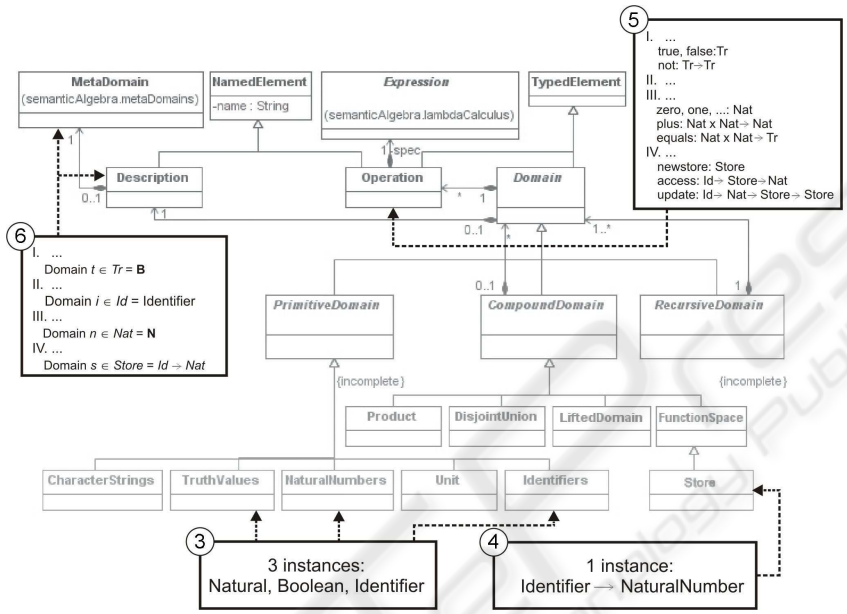

Figure 7: Instantiated elements in the semantic domains.

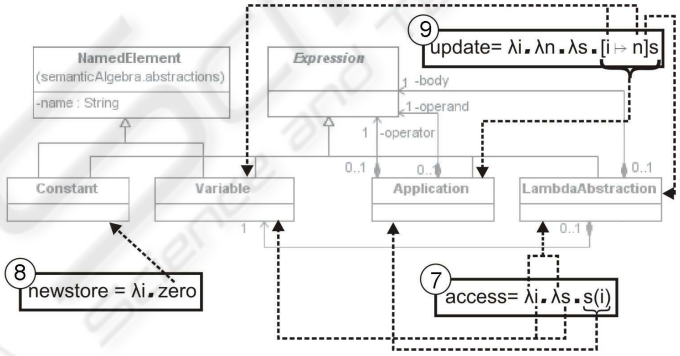

Figure 8: Three specifications built on the λ-calculus formalism.

s concluding the specification.

6 RELATED WORKS

There exists some proposed semantic metamodels in the context of formal semantics. However, no one focuses on the semantic algebra of the denotational formalism. Kent et al. (Kent et al., 1999) redefines metamodels for UML/OCL in order to incorporate formal semantics. In this case, the denotational approach is realized by redefining: (i) the semantic metamodel for the language of models (classes, roles, models); (ii) the semantic metamodel for the language of instances (objects, links); and (iii) the mapping between these two languages as semantic equations. However this work differs from ours because it suffers of the lack of formalization of the languages in the domain and codomain. The mapping is not enough to ensure conformance between models and makes hard to consider

extensions for dynamic aspects of behavior.

Concerning model transformations, Dominguez *et al.* (Dominguez et al.,) investigate the explicit correspondence between metamodels and ontologies of different languages. They consider refinement of metamodels instead of models, as we do here. By constructing a chain of metamodels, according to some refinement laws, they analyze refinements in both sides until that some convergence can be found to compare meanings. The proposed idea can help to delimit the difficulties in assuring that semantics are preserved in some translation but has the limitation of dealing with the language level, which requires a high-level of expertise.

7 CONCLUSIONS AND FUTURE WORK

We proposed a MOF metamodel for semantic algebra and implementation of its corresponding reductions. The former formally introduces domains and operations of programming languages, and the later simplify models towards a normal-form. They play an important role in the MDA framework because they: (i) put forward semantic algebra in the MDA vision, allowing specifications from or to this formalism; (ii) put forward semantic algebra as a DSL allowing formal semantic applications; (iii) enables reasoning about meaning and behavior in order to get a formal conformance between models by using reductions after transformations; and (iv) have potential to emerge as crucial elements in the MDA framework in order to allow semantic preservation in model transformations.

The proposed metamodel and reductions are part of an ongoing project whose main goal is to guarantee semantic preservation in model transformations in the MDA infrastructure. To that effect, the widely recognized potential of semantic algebra as a semantic notation for syntactic constructs of programming languages put its metamodel as a crucial element in the puzzle of MDA artifacts.

Although QVT is the current OMG's proposal for specifying transformations in the MDA vision, we have adopted ATL. The main reason for this choice is that QVT tools still have low robustness and its use is not widely disseminated. On the other hand, ATL is employed by an increasing and enthusiast community, with full support to model operations, where, in an integrated way, one can specify and instantiate metamodels as well as specifying and executing transformations on them.

As future work, we intend to propose an extension on the MDA four-layer architecture in order to include the semantic algebra metamodel as required element for guaranteeing semantic preservation in model transformations. The reductions will be inferred and performed automatically by the use of rewriting logic in rewrite systems.

REFERENCES

(2004). *ADT: Eclipse development tools for ATL*.

Böhm, C., editor (1975). *Lambda-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*. Springer.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop On Generative Techniques in the Context of the Model Driven Architecture*.

Dominguez, E., Rubio, A., and Zapata, M. Mapping models between different modeling languages. In *Workshop on Integration and. Transformation of UML Models, 2002*, pages 18–22.

Kent, S., Gaito, S., and Ross, N. (1999). A meta-model semantics for structural constraints in UML. In Kilov, H., Rumpe, B., and Simmonds, I., editors, *Behavioral specifications for businesses and systems*, chapter 9, pages 123–141. Kluwer Academic Publishers, Norwell, MA.

Kleppe, A. and Warmer, J. (2003). Do mda transformations preserve meaning? an investigation into preserving semantics. In *International Workshop on Metamodelling for MDA*.

OMG (2007). Model driven architecture. http://www.omg.org/mda/.

Schmidt, D. A. (1986). *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA.

Scott, D. and Strachey, C. (1971). Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*.