Formal MOF Metamodeling and Tool Support

Liliana Favre[1,2] and Daniel Duarte[1]
[1]*Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina*
[2]*Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina*

Keywords: Model Driven Development, MDA, Metamodeling, MOF, Formal Specification, ANTLR, Test Driven Development (TDD).

Abstract: Model Driven Development (MDD) has emerged as a new road to software development industrialization. The most well-known realization of MDD is the Model Driven Architecture (MDA). The essence of MDA is the metamodel MOF (Meta Object Facility) allowing interoperability of different kind of artifacts from multiple technologies. It is important to formalize and reason about MOF metamodels properly. In this paper, we propose a rigorous framework for reasoning about "correctness" of metamodels. Our main contribution is the integration of MOF metalanguage with formal specification languages based on the algebraic formalism. We define NEREUS, a formal metamodeling language, and processes for reasoning about MOF-like metamodels such as ECORE metamodels. The paper describes a set of tools developed to make formal metamodeling feasible in practice.

1 INTRODUCTION

In the last decade, Model Driven Development (MDD) has emerged as a new road to the software development industrialization (Brambilla et al., 2012). MDD refers to a range of development approaches based on the use of models as first class entities. The most well-known is the Object Management Group standard Model Driven Architecture (MDA), i.e., MDA is a realization of MDD (OMG, 2015) (MDA, 2014). Among the benefits provided by MDA, it can be remarked the improvement of interoperability, productivity, code and processes quality and, software evolution costs.

The key idea behind MDA is to separate the specification of the system functionality from its implementation on specific platforms, increasing the degree of automation and achieving interoperability with multiple platforms. Any artifact in MDA is a model and any MDA process is carried out as a sequence of model transformations. Model and model transformations need to be expressed in some notation and the MDA standard to express them is the MOF (Meta Object Facility) metamodel. It can be considered the essence of MDA allowing different kinds of artifacts from multiple technologies to be used together in an interoperable way (MOF, 2015) (MOF, 2006). MOF provides two metamodel: EMOF (Essential MOF) and CMOF (Complete MOF). The former favors simplicity of implementation over expressiveness, while the latter is more expressive, but more complex. Based on MOF transformations, the MDA unifies every step of software development.

The Eclipse Modeling Framework (EMF) has become the reference platform for developing MDD tools. Particularly, its meta-metamodel ECORE is an implementation of MOF (Steinberg et al., 2008).

It is important to formalize and reason about MOF metamodels and we propose to exploit the strong background achieved by the community of formal methods. In previous work, we presented NEREUS, a formal language for metamodeling that combines the most successful features of algebraic languages explored in different contexts (Favre, 2009). It can be viewed as a concrete syntax for MOF extended with additional properties expressed by axioms. Besides, NEREUS is an intermediate notation that can be integrated with property-oriented formal approaches. Particularly, we integrate NEREUS with the Common Algebraic Specification Language (CASL) as target algebraic language (Bidoit and Mosses, 2004).

Our current contribution can be viewed as an evolution of the previous results. We describe practical and theoretical advances. From the theoretical point of view, we describe the current syntax of NEREUS and its semantics that was given by translating it to CASL. On the practical point of

Favre, L. and Duarte, D.
Formal MOF Metamodeling and Tool Support.
DOI: 10.5220/0005689200990110
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 99-110
ISBN: 978-989-758-168-7

view, the current version of an analyzer of NEREUS and a translator from NEREUS to CASL will be described.

The rest of the paper has the following structure. Section 2 introduces related work and Section 3 presents motivation remarking our contribution. Section 4 introduces the features of the NEREUS language. Section 5 describes tools developed to assist in formal metamodeling processes. Finally, in Section 6 we present conclusions.

2 RELATED WORK

We describe related work to formalization of MOF-like metamodels.

The state of the art and emerging research challenges for metamodeling are described in (Sprinkle et al., 2010). Authors review approaches, abstractions, and tools for metamodeling, evaluate them with respect to their expressive power, investigate what role(s) metamodels may play at run-time and how semantics can be assigned to metamodels and the domain-specific modeling languages they could define. They also highlight emerging challenges regarding the management of complexity, consistency, and evolution of metamodels, and how the semantics of metamodels will impact on each of them.

The MetaModeling Language (MML) is a subset of UML that was proposed as a core language to define UML (Clark et al., 2001). It has a formal semantic based on a small language called MML calculus. It is an imperative object-oriented calculus, which captures the essential operational features of MML and is inspired in the calculus proposed by (Cardelli and Abadi, 1991).

Varró and Pataricza (2003) presented a visual and formally precise metamodeling (VPM) framework that is capable of uniformly handling arbitrary models from engineering and mathematical domains. They propose a multilevel metamodeling technique with precise static and dynamic semantics (based on a refinement calculus and graph transformation) where the structure and operational semantics of mathematical models can be defined in a UML notation.

A graph grammar to generate instances of metamodels, one of the limitations of metamodel implementations, for instance in Eclipse Modeling Framework is described in (Erigh et al., 2006). An instance generating graph grammars for creating instances of a metamodel was introduced in (Karsten et al., 2006).

The correspondence semantic between UML class diagrams and Alloy is described in (Anastasakis et al., 2007). Alloy is a modeling language based on first order relational logic. Its analyzer is equipped with a SAT-based engine that can be used to generate valid system configurations or counterexamples to a property.

Boronat and Messeguer (2010) describe an algebraic, reflexive and executable framework for metamodeling in MDD. The framework provides a formal semantic of the notions of metamodel, model and conformance relation between a model and a metamodel. The semantic is integrated to EMF as a plugin called MOMENT (MOdelmanageMENT). The underlying formalism of MOMENT is MAUDE. Bridges between technological spaces MAUDE and EMF that provide interoperability were defined.

The problem of identifying, predicting and evaluating the significance of the metamodel change impact over the existing artifacts is described in (Iovino, Pieroantonio and Malavolta, 2012). The approach is based on the concept of megamodel. In this context a megamodel is considered a model of which at least some elements represent and /or refer to models and metamodels. This approach allows developers both, to establish relationships between the metamodel and its related artifacts, and to automatically identify those elements within the various artifacts affected by the metamodel changes.

Barbier et al., (2013) describe how to construct metamodels based on the constructive logic, along with inherent proofs. The key contribution is a generative approach to construct new metaclasses from existing ones. Authors propose to define the entire MOF in this way and implement it in Coq Proof Assistant (https://coq.inria.fr/). They do not target automatic transformation from Coq to MOF-like metamodels.

An approach to the metamodel formalization based on algebraic data types and constraint logic programming (CLP) is described in (Jackson et al., 2011). Proofs and test-case generation are encoded as CLP satisfiability problems to automatically be solved. Authors describe the framework Formula to solve proofs to verify properties of the metamodels that are viewed as instances of CLP. The Eclipse plugin CD2FORMULA that implements in a way aligned with MDA the translation of UML class diagrams to FORMULA is described in (Perez and Porres, 2014). The proposed framework can be used to reason, validate and verify UML software designs by checking correctness properties and generating model instances by using a model exploration tool based on Formula.

3 MOTIVATION

MOF metamodels are specified by using restricted UML class diagrams and annotations OCL. On the one hand, UML has the advantage of visualizing language constructs. On the other hand, OCL has a denotational semantics that has been implemented in tools allowing dynamic validation of snapshots.

It is important to formalize and reason about MOF metamodels properly. Instantiating a metamodel produces models, which in turn are instantiated. Having errors in a metamodel leads to having errors in its model instances. Besides, a model can be well-formed but still be incorrect. A combination of MOF metamodeling and formal specification can help metadesigners to address these issues.

A formal specification technique must at least provide syntax, some semantics and an inference system. The syntax defines the structure of the text of a formal specification including properties that are expressed as axioms (formulas of some logic). The semantics describes the models linked to a given specification; in the formal specification context, a model is a mathematical object that defines behavior of the realizations of the specification. The inference system allows defining deductions that can be made from a formal specification. These deductions allow new formulas to be derived and checked. So, the inference system can help to automate testing, prototyping or verification.

Current metamodeling tools enable code generation and detect invalid constraints however, they do not find instances of the metalanguage (models). This is a limitation for certain applications related with MDA. For instance, to have enough valid instances available is a requisite to test model transformations.

Our main contribution is related to the integration of specifications expressed in MOF metalanguage with formal specification languages, based on the algebraic formalism. We define the NEREUS language, in particular. It is a formal notation closed to MOF metamodels that allows metadesigners who must manipulate metamodels to understand their formal specification. The semantic of MOF metamodels (that is specified in OCL) can be enriched and refined by integrating it with NEREUS. This integration facilitates proofs and test of models and model transformations via the formal specification of metamodels.

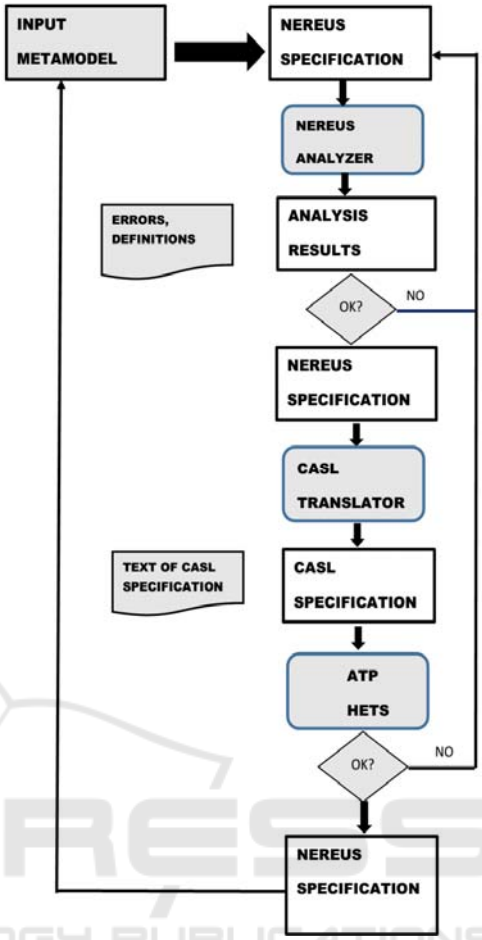


Figure 1: Our contribution: Typical flow with formal tools.

Figure 1 summarizes our approach. First, a specification of a MOF metamodel is transformed into a NEREUS specification. A system of transformation rules to automatically transform MOF into NEREUS was previously described (Favre, 2009). Next, the formal specification is analyzed by using the analyzer of NEREUS and is modified according to the results of the translation process with the goal of obtaining a syntactically correct specification. Subsequently, the NEREUS specification is translated to a CASL specification by using a NEREUS-to-CASL Translator. NEREUS could be linked through CASL with Automatic Theorem Provers (ATP) provided by the Heterogeneous Tool SET (HETS) (Hets, 2015) (Mossakowski et al., 2014). ATPs allow performing a consistency analysis of the metamodel and achieving an analyzed specification. The initial MOF specification can be improved by reinjecting the changes introduced in the latter NEREUS specification.

In this paper, the emphasis is given to the NEREUS formalization of MOF metamodels and the tool support for it.

4 FORMALIZING METAMODELS

The MOF modeling concepts are "classes, which model MOF meta-objects; associations, which model binary relations between meta-objects; Data Types, which model other data; and Packages, which modularize the models" (MOF, 2006 pp. 2-6). OCL can be used to attach consistency rules to metamodel components.

The MOF model is self-describing, that is to say it is formally defined using its own metamodeling constructs. This provides a uniform semantic treatment between artifacts that represent models and metamodels in MDA.

In this section we provide a general background of the NEREUS language that allows specifying MOF and ECORE metamodels. NEREUS provides modeling concepts that are supported by MOF and the UML Infrastructure, including classes, associations and packages and, mechanisms for structuring them. First, we describe the syntax of classes, associations and packages. Next, we present examples of NEREUS specifications (section 4.2). Finally, in 4.3 we analyze why to use NEREUS.

4.1 NEREUS Syntax

4.1.1 Defining Classes

Classes may declare types, attributes, operations and axioms which are formulas of first-order logic. They are structured by different kinds of relations: importing, inheritance, subtyping and associations. Next, we show the syntax of a class in NEREUS:

CLASS className [<parameterList>]
IMPORTS <importsList>
IS-SUBTYPE-OF <subtypeList>
INHERITS <inheritsList>
ASSOCIATES <associatesList>>
BASIC CONSTRUCTOR(S) <constructorList>
DEFERRED
TYPE(S) <sortList>
ATTRIBUTE(S) <attributeList>
OPERATION(S) <operationList>
EFFECTIVE
TYPE(S) <sortList>
ATTRIBUTE(S) <attributeList>
OPERATION(S) <operationList>
AXIOMS <varList>
<axiomList>
END-CLASS

NEREUS distinguishes variable parts in a specification by means of explicit parameterization. The elements of <*parameterList*> are pairs *C1:C2* where *C1* is the formal generic parameter constrained by an existing class *C2* (only subclasses of *C2* will be actual parameters). In particular, the binding *C1:ANY* expresses a parameterization without restrictions and can be denoted by *C1*. The IMPORTS clause expresses client relations. The specification of the new class is based on the imported specifications declared in <*importList*> and their public operations may be used in the new specification.

NEREUS distinguishes inheritance from subtyping. Subtyping is like inheritance of behavior, while inheritance relies on the module viewpoint of classes. Inheritance is expressed in the INHERITS clause; the specification of the class is built from the union of the specifications of the classes appearing in the <*inheritsList*>. Subtypings are declared in the IS-SUBTYPE-OF clause. A notion closely related with subtyping is polymorphism. NEREUS allows us to define local instances of a class by the following syntax *ClassName [rename <bindingList>]* where the elements of <*bindingList*> can be pairs of identifiers *nameTo* as *nameFrom* separated by comma.

The BASIC CONSTRUCTORS clause lists the operations that are basic constructors of the interest type. NEREUS distinguishes deferred and effective parts. The DEFERRED clause declares new types, attributes or operations that are incompletely defined. The EFFECTIVE clause declares types, attributes and operations completely defined.

The ATTRIBUTES clause introduces, like MOF, an attribute with the following properties: name, type, multiplicity specification and "*isDerived"* flag. OPERATIONS clause introduces the operation signatures, the list of their arguments and result types. An attribute or parameter may be optional-value, single value, or multi-valued depending on its multiplicity specification. The multiplicity syntax is aligned with the MOF syntax.

Operations can be declared as total or partial. Partial functions must specify its domain by means of the PRE clause that indicates what conditions the function´s arguments must satisfy to belong to the function's domain. NEREUS allows us to specify operation signatures in an incomplete way. NEREUS supports higher-order operations (a function f is higher-order if functional sorts appear in a parameter sort or the result sort of f). In the context of OCL

Collection formalization, second-order operations are required but NEREUS support higher-order.

In NEREUS it is possible to specify any of the three levels of visibility for operations (public, protected and private) and incomplete functionalities denoted by underscore in the operation signature.

4.1.2 Defining Associations

NEREUS provides a component Association, a taxonomy of constructor types, that classifies binary associations according to kind (aggregation, composition, ordinary association), degree (unary, binary), navigability (unidirectional, bidirectional) and, connectivity (one-to-one, one-to-many, many-to-many) (Favre, 2009).

The component Association provides Relation Schemes that can be used in the definition of concrete associations by instantiating classes, roles, visibility, and multiplicity. Associations can be restricted by using static constraints in first order logic. New associations can be defined by the ASSOCIATION construction. The IS clause expresses the instantiation of <*typeConstructorName*> with classes, roles, visibility, and multiplicity. The CONSTRAINED-BY clause allows the specification of static constraints in first order logic. Next, we show the association syntax:

ASSOCIATION <relationName>
IS <typeConstructorName>
[…:class1;…:class2;…:role1;…:role2;…:mult1;…:mult2;
…:visibility1;…:visibility2]
CONSTRAINED-BY <constraintList>
END-ASSOCIATION

Associations are defined in a class by means of the ASSOCIATES clause:

CLASS className…
ASSOCIATES <<associationName>>

4.1.3 Defining Packages

The package is the mechanism provided by NEREUS for grouping related model elements together in order to manage complexity and facilitate reuse. Like MOF, NEREUS provides mechanisms for metamodel composition and reuse. The IMPORTING clause lists the imported packages; the GENERALIZATION clause lists the inherited packages; NESTING clause lists the nested packages and CLUSTERING clause list the clustering ones. Classes, associations and packages can be <*elements*> of a package. The package has the following NEREUS syntax:

PACKAGE packageName
IMPORTING <importsList>
GENERALIZATION <inheritsList>
NESTING <nestingList>
CLUSTERING <clusteringList>
<elements>
END-PACKAGE

4.1.4 Examples

Following, we show by examples the syntax of NEREUS. First, we show partially a Collection specification.

CLASS Collection [Elem]
BASIC CONSTRUCTORS create, add
DEFERRED
TYPE Collection
OPERATIONS
create :-> Collection;
add : Collection * Elem -> Collection;
count : Collection * Elem ->Integer;
collect : Collection * (Elem ->Elem1: ANY)-> Collection;
EFFECTIVE
OPERATIONS
isEmpty : Collection ->Boolean;
size: Collection ->Integer;
includes : Collection * Elem ->Boolean;
excludes : Collection * Elem ->Boolean;
includesAll : Collection * Collection ->Boolean;
excludesAll : Collection * Collection ->Boolean;
forAll : Collection * (Elem ->Boolean) -> Boolean;
exists : Collection * (Elem ->Boolean) ->Boolean;
select : Collection * (Elem ->Boolean) ->Collection;
reject : Collection * (Elem ->Boolean) ->Collection;
iterate : Collection *
((Elem * Acc: ANY)->Acc: ANY) *
(->Acc:ANY) ->Acc: ANY ;

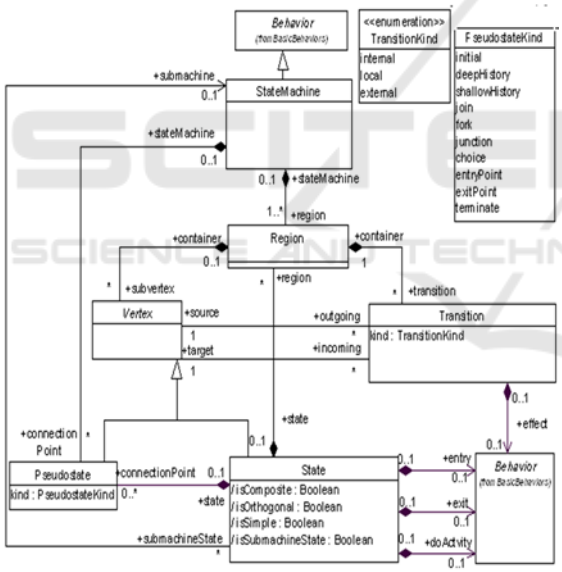
AXIOMS c, c1: Collection; e,e1: Elem;
f: Elem ->Boolean;
g: Elem * Acc -> Acc;
base: -> Acc;
isEmpty(c) = (size(c) = 0);
iterate (create(), g, base()) = base();
iterate (add (c, e), g, base()) =
g(e, iterate(c, g, base()));
size(create()) = 0;
size(add(c, e)) = 1 + size(c);
includes(create(), e) = False;
includes(add(c, e), e1) =
if e = e1 **then** True **else** includes(c, e1) **endif** ;
forall(create(), f) = True;
forall(add(c, e), f) = f(e) and forAll(c, f);
exists (create(), f) = False;
exists (add (c, e), f) = f(e) or exists(c, f);
select(create(), f) = create();
select (add (c, e), f) = **if** f(e) **then** add(select(c, f), e) **else**
select(c, f) **endif** ;
…
END-CLASS

Next, we show the formalization of a simplified package: *StateMachineMetamodel*. Figure 2 depicts a simplified diagram. A behavior *StateMachine* comprises one or more *Regions*, each *Region* containing a graph (possibly hierarchical) comprising a set of *Vertex* interconnected by arcs representing *Transitions*. *StateMachine* execution is triggered by appropriate *Event* occurrences. OCL can be used to constraint components of the metamodel.

PACKAGE StateDiagramMetamodel
IMPORTING TransitionKind, PseudoStateKind

CLASS StateMachine
IS-SUBTYPE-OF
UML::CommonBehaviors::BasicBehaviors::
Behavior
ASSOCIATES
<<StateMachine-State>>
<<StateMachine-PseudoState>>
<<StateMachine-Region>>

AXIOMS a: <<StateMachine-PseudoState>>;
sm: StateMachine;



Context Statemachine
conectionPoint->
forAll (c | c.kind = Pseudostate::entryPoint or
c.kind = Pseudostate::exitPoint)
Context PseudoState
(self.kind = Pseudostate::initial) implies
(self.outgoing->size <= 1)
Context Region
self.subvertex-> select (v| v.oclIsKindOf
(Pseudostate))- >
select (p: Pseudostate|p.kind = Pseudostate::initial)->
size () <= 1

Figure 2: The StateMachine Metamodel.

/* *The connection points of a state machine are pseudostates of kind entry point or exit point**/
forAll (c) (get_connectionPoint (a, sm), [(kind(c) = PseudoState:: entryPoint or (kind(c) = PseudoState:: exitPoint]);
END-CLASS

CLASS Region
IS-SUBTYPE-OF UML::Classes::Kernel::Namespace
ASSOCIATES
<<State-Region>>
<<StateMachine-Region>>
<<Region-Vertex >>…
AXIOMS a: <<Region-Vertex>>; r: Region;
/**A region can have at most one initial vertex**/
size (select (p) (select (v) (get_subvertex (a, r),
[oclIsKinfOf (v, PseudoState)])
[kind(p) = PseudoState::initial()])) <= 1;
END-CLASS

CLASS PseudoState
IMPORTS PseudoStateKind
IS-SUBTYPE-OF Vertex,
UML::Classes::Kernel::NameElement
ASSOCIATES
<<Vertex-Transition_1>>
<<Vertex-Transition_2>>
<<StateMachine-PseudoState>> ...
EFFECTIVE
OPERATION
kind: PseudoState -> PseudoStateKind;
AXIOMS ps: PseudoState; a: Vertex-Transition-1
/**An initial vertex can have at most one ongoing transition**/
kind (ps) = Pseudostate::initial implies
size (get_outgoing (a,ps)) <=1;
END-CLASS

ASSOCIATION stateMachine-PseudoState
IS Composition_2 [StateMachine: class1; PseudoState: class2; stateMachine: role1; conectionPoint: role2; 0..1: mult1; *: mult2; +: visibility1;+: visibility2]
CONSTRAINED-BY StateMachine: subsets namespace, PseudoState: subsets ownedMember
END-ASSOCIATION

ASSOCIATION Vertex-Transition-1
IS Bidirectional [Vertex: class1; transition: class2; source: role1; outgoing: role2; 1: mult1; *: mult2; +: visibility1; +: visibility2]
END-ASSOCIATION

ASSOCIATION Region-Vertex
IS Composition_2 [Region: class1; Vertex: class2; container: role1; subvertex: role2; 0..1: mult1; *: mult2; +: visibility1; +: visibility2]
END-ASSOCIATION
...
END-PACKAGE

4.2 NEREUS Semantic

The semantics of NEREUS was constructively given by translation to CASL. CASL is an algebraic language based on a critical selection of known constructs such as subsorts, partial functions, first-order logic, and structured and architectural specifications.

We select CASL due to it is at the center of a family of specification languages, is supported by tools and facilitates interoperability of prototyping and verification tools. CASL is linked to ATP through HETS. It is worth considering that HETS is a parsing, static analysis and proof management tool combining various such tools for individual specification languages, thus providing a tool for heterogeneous multi-logic specification. HETS is based on a graph of logics and languages (formalized as so-called institutions), their tools, and their translations. This provides a clean semantics of heterogeneous specification, as well as a corresponding proof calculus. However, CASL syntax is far from the way of specifying of metadesigners.

We define a way to automatically translate each NEREUS construct into CASL, including classes, different kinds of relations and packages (Favre, 2009).

Next, we describe the most interesting problem in the translation, how to translate associations due to algebraic languages do not follow the MOF structuring mechanisms. The graph structure of a class diagram involves cycles such as those created by bidirectional associations. However, the algebraic specifications are structured hierarchically and cyclic import structures between two specifications are avoided. An association in UML can be viewed as a local part of an object and this interpretation cannot be mapped to classical algebraic specifications which do not admit cyclic import relations. 1

We propose an algebraic specification that considers associations belonging to the environment in which an actual instance of the class is embedded. Let *Assoc* be a bidirectional association between two classes called *Asource* and *Bsource* the following steps can be distinguished in the translation process:

Step1: Regroup the operations of classes *Asource* and *Bsource* distinguishing operations local to *Asource*, local to *Bsource* and, local to *Asource* and *Bsource* and *Assoc.*

Step 2: Construct the specifications *A* and *B* from *Asource* and *Bsource* where *A* and *B* include local operations to *Asource* and *Bsource* respectively.

Step 3: Construct specifications *Collection[A]* and *Collection [B]* by instantiating reusable schemes.

Step 4: Construct a specification *Assoc* (with *A* and *B*) by instantiating reusable schemes in the component *Association*.

Step 5: Construct the specification *A&B* by extending *Assoc* with *A, B* and the operations local to *A*, *B* and *Assoc*.

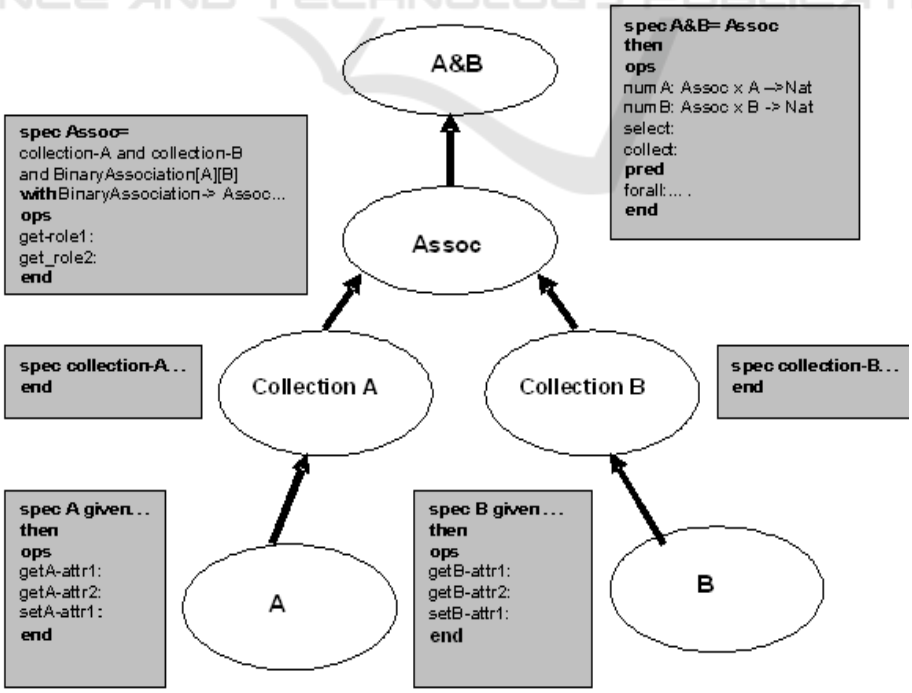


Figure 3: From NEREUS to CASL: Translating Associations.

Another interesting problem is how to translate higher order operations into first-order in CASL. The classes that include higher order operations are translated inside parameterized first-order specifications. The main difference between higher order specifications and parameterized ones is that, in the first approach, several function-calls can be done with the same specification and parameterized specifications require the construction of several instantiations.

Next, we show the translation of the Collection specification shown in 4.1.4 to CASL. Take into account that there are as much functions *f1*, *f2*, *f3*, and f4 as functions *select*, *reject*, *forAll* and *exists*. There are as much functions *base* and *g* as functions iterate too.

spec Operation [sort X] =
Z_1 and Z_2 and ... Z_r
then
preds
$f1_j$: X; | 1 ≤ j ≤ m
$f2_j$: X ; | 1 ≤ j ≤ n
$f3_j$: X; | 1 ≤ j ≤ k
$f4_j$: X | 1 ≤ j ≤ l
ops
$base_j$: -> Z_j ; | 1 ≤ j ≤ r
g_j: Z_j x X -> Z_j | 1 ≤ j ≤ r
end

spec Collection [sort elem]
given NATURAL-ARITHMETIC=
Operation [elem]
then
generated type
Collection ::= create | add (Collection ; elem)
preds
isEmpty : Collection;
includes: Collection * elem;
includesAll: Collection * Collection;
$forAll_i$: Collection; |1 ≤ i ≤ k
$exists_i$: Collection |1 ≤ i ≤ l
ops
size: Collection -> Nat;
$iterate_i$: Collection -> Z_j ; | 1≤ i ≤ r
$select_i$: Collection -> Collection; |1≤i ≤ m
$reject_i$: Collection -> Collection; |1≤i ≤ n
…
forall c, c1: Collection; e,e1: elem

- isEmpty (create)
- includes (add (c, e), e1) < = >
  (e = e1) \/ includes(c,e1))
- includesAll (c, add (c1, e)) =
  includes(c, e) /\ includesAll (c, c1)
- $forAll_i$ (add(c,e)) < = >
  $f3_i$ (e) /\ $forAll_i$ (c) |1 ≤ i ≤ k
- $exists_i$ (add(c,e) < = >
  $f4_i$(e) \/ $exists_i$ (c) |1 ≤ i ≤ l
- $select_i$ (create) = create
- $f1_i$(e) = > $select_i$ (add (c, e)) =
  add ($select_i$ (c), e) ; | 1 ≤ j ≤ m
- ¬ $f1_i$ (e) = >
  $select_i$ (add (c, e)) = $select_i$ (c) |1≤i ≤ m

…

4.3 Why to Use NEREUS?

Such as MOF is a DSL (Domain Specific Language) to define semi-formal metamodels, NEREUS can be viewed as a DSL for defining formal metamodels.

Advantage of our approach is linked to pragmatic aspects. NEREUS is a formal notation closed to core concepts of MOF metamodels that allows metadesigners who must manipulate metamodels to understand their formal specification.

NEREUS is a metamodeling formal language with strong abstraction from details of the classical mathematical notation of algebraic languages. In comparison to CASL (or other formal languages) it may use metamodel constructs, be easier to use and may automate significant issues of the metamodel specification (e.g. association specification) making the process of developing a formal specification simpler and more understandable relative to "lower level" formal languages. The mathematics of NEREUS specification is easily learned and used supporting other way of expressing metamodels giving metadesigners a better understanding early on them.

A metadesigner can reflect exactly the MOF constructs in NEREUS delegating the translation of them to a translator that automatize the process. For instance NEREUS-to-CASL translator translates automatically NEREUS associations into CASL starting from the bases described in section 4.2.

Another important issue is that NEREUS, like MOF, provides mechanisms to structure large specifications in order to be legible and understandable. NEREUS provides a set of features which allow the modularization of specifications. However, a minimum knowledge about algebraic specifications or about the semantics of NEREUS expressions is a requisite for metadesigners.

5 TOOLS FOR NEREUS

Our approach provides an appropriate set of tools to make formal metamodeling feasible in practice. In this section we describe them:

- A parser for NEREUS which includes lexical, syntactic and semantic analysis. It was developed in ANTLR 4 for Java. ANTLR (*Another Tool for Language Recognition*) is a powerful parser

generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From an additional grammar, ANTLR generates a parser that can build and walk parse trees (Parr, 2013).

- A translator of NEREUS specifications into CASL specifications, developed in Java, that uses tree walkers generated automatically by ANTLR 4. It can be used to visit their nodes to execute application-specific code. It is worth considering that ANTLR 4 allows writing grammars specially designed for searching and processing syntax trees "on the fly", separating the parsing, search and process of structures. The translator from NEREUS to CASL is based on the constructive semantic described previously in 4.2.
- An application that provides the ability to write specifications NEREUS, integrating the analyzer and translator. The application is an IDE-style where the metadesigner is not only able to enter NEREUS text but see the result of its syntactic and semantic analysis. Another important output is the CASL text.

With regard to the generation of Java code for analyzers, it is sufficient to use ANTLR, however we decide to integrate it with the ANTLRWorks application that makes use of ANTLR and provides a comfortable and appropriate interface for writing and debugging grammars through an intuitive and easy graphical interface.

Both the Java language and the tool ANTLR are open source, providing the ability to use them without major technological or economic constraints, allowing access to details of their implementations. The fact that ANTLR was implemented in Java provides easy integration with the application developed to achieve the final product that provides reusable applications across multiple desktop platforms, which are usually used by members of the development or design teams.

The development process for generating the lexical analyzer, parser and semantic analyzer was TDD (Test Driven Development). It is a software development process that relies on the repetition of small steps: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

Another aspect to note is that while the automatic testing was based on the JUnit tool for Java, a proper testing engine was implemented. In this engine, each test is an example of NEREUS text and a set of directives that indicate what are the results expected from the analyzer.

Figure 4 shows two screenshot. The first one shows the translation from NEREUS to CASL of a simple class; the second one is a screenshot of the main application screen depicting the main panels.

In the main part of the screen (Figure 4), we can see the edition panel of NEREUS specifications. It has the common characteristics of code editors, i.e., syntax highlighting, line numbers and highlighting of the current line among others.

Immediately below, the panel of errors can be seen. It indicates errors showing their type (lexical errors, syntactic, semantic errors, or general errors), its location in the text (line number and column) and the corresponding messages. Additionally it is possible to position the cursor on errors, making double-click on them. This panel has also a checkbox "Automatic Analysis" which, if marked, enables re-analyze the text of each new edition of NEREUS showing the updated results.

At the top of the application there is a menu bar and a toolbar with buttons, both with general functionality of NEREUS files (new file, existing open, save). In particular, it included the option for the classpath edition of the NEREUS specification, which is located in the Options menu. Similar to the way in which Java performs the search of classes, the analyzer will seek NEREUS specifications (Classes, packages, association, relation scheme) that are referenced within the directories in the classpath.

On the right, there is a panel of multiple functions with different tabs. They provide information about the test result: general information (General), the syntax tree (Syntax Tree), the tree of items (NEREUS Tree), detail of the statements found during the edition of a class (declarations that are only available for classes and relation schemes) and CASL text generated from the specification NEREUS (CASL).

6 CONCLUSIONS

We have provided a metamodeling framework based on MOF and the algebraic formalism that focus on automatic proofs and tests. The central components of our approach are the definition of the algebraic language NEREUS and the development of tools for formal metamodeling: the NEREUS analyzer and the NEREUS-to-CASL translator.

With respect to NEREUS, our approach focuses on interoperability of formal languages. Considering that there exist many formal algebraic languages,

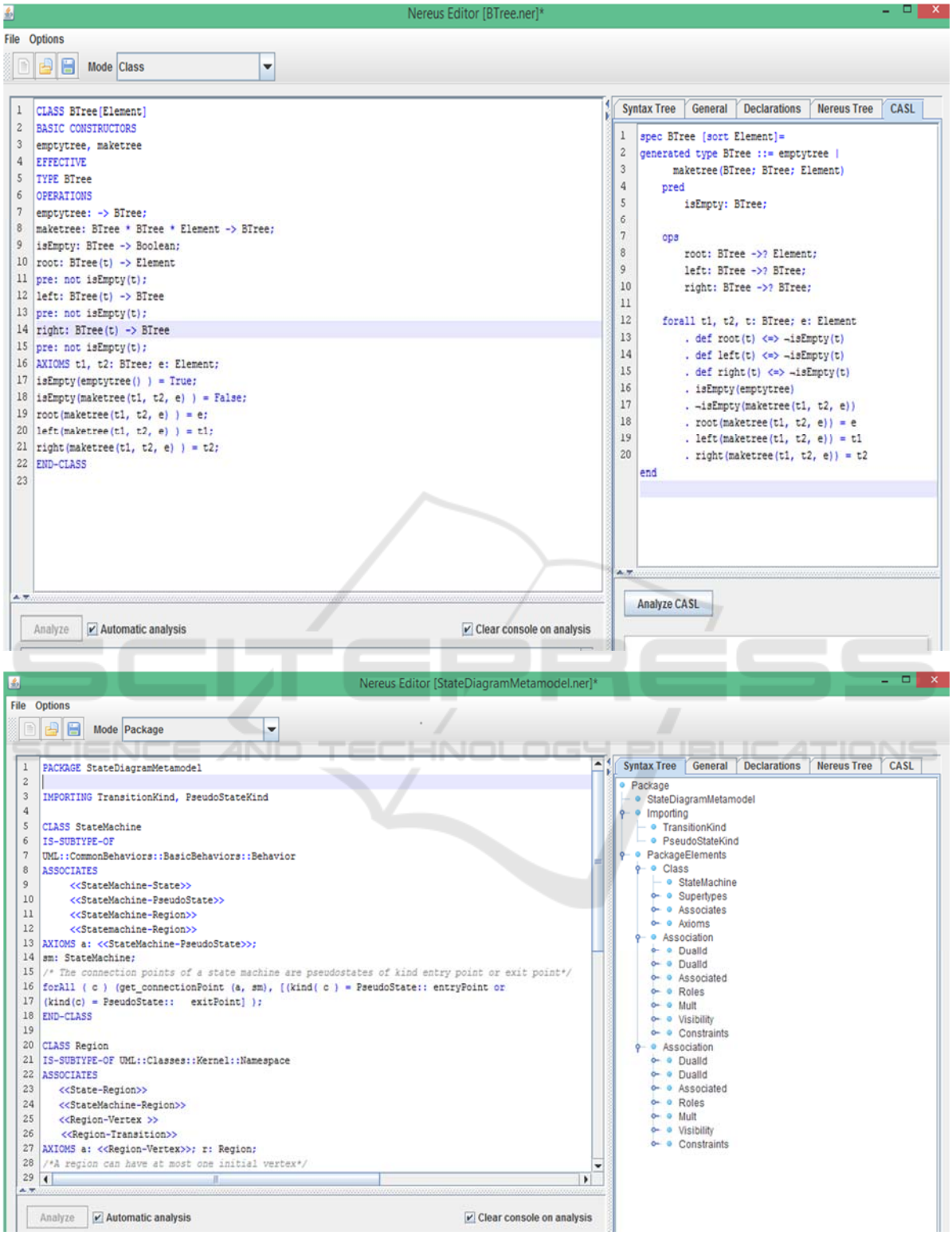


Figure 4: The NEREUS Analyzer.

NEREUS allows any number of source languages such as different Domain Specific Language (DSLs) and target languages (different formal language) could be connected. Such as MOF is a DSL to define semi-formal metamodels, NEREUS can be viewed as a DSL for defining formal metamodels. Another advantage of our approach is linked to pragmatic aspects. NEREUS is a formal notation closed to MOF metamodels.

With respect to the NEREUS-to-CASL translator, due to the NEREUS semantics was given for translation to CASL, we develop a translator that could be integrated with HETS and different ATP.

Other work shows the integration of NEREUS with MOF (Favre, 2009). MOF metamodels can be transformed into NEREUS specifications to be validate and verify and changes reinjected into MOF metamodels. A system of transformation rules to automatically transform MOF into NEREUS, was previously described in (Favre, 2009). Our approach allows translating MOF into NEREUS integrating OCL with NEREUS and facilitating the translation process from MOF.

Rather than requiring developers to manipulate formal specifications, the idea is to provide rigorous foundations for MOF-like metamodels in order to develop tools that, on the one hand, take advantage of the power of formal languages and, on the other hand, allow developers directly manipulating MDA models.

How to perform testing and verification on a large scale is still a challenge. We also consider interesting another challenge: to analyze issues related to evolution models driven by metamodel evolution in an incremental verification way.

REFERENCES

Anastasakis, K., Bordbar, B., Georg, G., Ray, I., 2007. UML2Alloy: A Challenging Model Transformation, In *Proceedings Model Driven Engineering Languages and Systems (Models 2007),* Lecture Notes in Computer Science 4735, Heidelberg: Springer-Verlag, pp. 436-450.

Barbier, P., Casteran, E., Cariou, E., le Goaer, O., 2013. Adaptative software based on correct-by construction metamodels, Chapter 13. In *Progressions and Innovations in Model Driven Software Engineering,* Hershey, PA: IGI Global, pp. 308-325.

Bidoit, M., Mosses, P., 2004. *CASL User Manual Introduction to Using the Common Algebraic Specification Language.* Lecture Notes in Computer Science 2900, Heidelberg: Springer-Verlag.

Boronat, A., Meseguer, J., 2010. An algebraic semantics for MOF. Formal Aspect of Computing 22, pp. 269-296.

Brambilla, M., Cabot, J., Wimmer, M., 2012. *Model-Driven Software Engineering in Practice.* USA: Morgan & Claypool.

Cardelli, L., Abadi, M., 1996. *A Theory of Objects*. Heidelberg: Springer-Verlag.

Clark, T., Evans,A., Kent, S. 2001. The Metamodeling Language Calculus: Foundation Semantic for UML, In *Proceedings of FASE 2001*, pp.17-31.

Duarte, D., 2015. Development of Formal Metamodeling Tools. System Engineer Thesis (L. Favre Supervisor). Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina.

EMF, 2015. Eclipse Modeling Framework, www.eclipse.org

Erigh, H., Erigh, K., Prange,U., Taentzer, G., 2006. Fundamentals of Algebraic Graph Transformation. *Monographs in Theoretical Computer Science.* EATCS Series. Springer-Verlag.

Karsten, E., Jochen, M., Kuster, G., Taentzer, J., 2006. Generating Instance Models from MetaModels. In *Winkelmann FMOODS 2006*, Lecture Notes in Computer Science 4037, Heidelberg: Springer-Verlag, pp. 156-170.

Favre, L., 2009. A Formal Foundation for Metamodeling, In *Ada-Europe 2009:* Lecture Notes in Computer Science 5570, Heidelberg: Springer-Verlag, pp. 177-191.

Hets, 2015. Heterogeneous Tool Set. www.informatik.uni bremen.de/agbkb/forschung/formal_methods/CoFI/hets/

Iovino, L., Pieroantonio, A., Malavolta, I., 2012. On the impact significance of metamodel evolution in MDE, *Journal of Object Technology* 11 (3), pp.1-33.

Jackson,E.K., Levendovszky, T., Balasubramanian, D., 2011. Reasoning about Metamodeling with Formal Specifications and Automatic Proofs, In *Proceedings Model Driven Engineering Languages and Systems Models 2011*, Lecture Notes in Computer Science 6981, Heidelberg: Springer-Verlag, pp. 653-667.

Jouault, F., Bézivin, J., 2006. KM3: a DSL for Metamodel Specification, Formal Methods for Open Object-Based Distributed Systems, Heidelberg: Springer-Verlag, pp. 171-185.

MDA, 2014. Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0, OMG Document ormsc/2014-06-01

MOF, 2015. OMG Meta Object Facility Core Specification, version 2.5, Document formal/2015-06-05 http://www. omg.org/spec/MOF/2.5

MOF, 2006. OMG Meta Object Facility (MOF) Core Specification, version 1.0.

Mossakowski, T., Maeder, C., Codescu, M., 2014. Hets User Guide, version 0.99, http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/

OCL, 2014. Omg Object Constraint Language (OCL), version 2.4," formal/2014-02-03, www.omg.org/ocl/ 2.4

Parr, T., 2013. *The Definitive ANTLR 4 Reference* (1st ed.), Pragmatic Bookshelf.

OMG, 2015. Object Mangement Group, www.omg.org

Pérez, B., Porres, I., 2014. An Overall Framework for Reasoning about UML/OCL Models Based on Constraint Logic Programming and MDA.

International Journal on Advances in Software, vol 7 no 1 & 2, http://www.iariajournals.org/software/

Steinberg, D., Budinsky, F., Paternostro , M., and Merks, E., 2008. EMF: Eclipse Modeling Framework, 2 ed. Addison-Wesley, Boston, MA

Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G. 2010. Metamodelling: State of the Art and Research Challenges. H. Giese et al. (Eds.), Lecture Notes in Computer Science 6100, Heidelberg, Springer-Verlag, pp. 57-76.

Varro, V., Pataricza, A., 2003. VPM: A visual, precise and multilevelmetamodeling framework for describing mathematical domains and UML. Journal of Software and System Modeling, 2 (3), pp. 187-210.