Teaching Formal Methods in the Context of Model Driven Engineering

Liliana Favre
liliana.favre@gmail.com
Universidad Nacional del Centro
de la Provincia de Buenos Aires
CICPBA
Tandil, Argentina

Abstract

The teaching of formal specifications plays a crucial role in the training of software engineering future professionals. Software failures can be detected in two major phases of the software development process: specification or modeling and programming. Taking into account that current students will not generate code in the conventional programming style, the verification and validation activities will also need to change significantly. In this paper, we propose to introduce formal methods in the context of Model Driven Engineering. Its underlying principles can be summarized as follows: all artifacts involved in a process can be viewed as models that conform to a particular metamodel, the process itself can be viewed as a sequence of model transformations and, all extracted information is represented in a standard way through metamodels. Model Driven Engineering will impact on software engineering curriculum, in both the body of knowledge and the structure of its courses. In light of this, metamodeling will have to be addressed from different visions. We propose to integrate metamodeling with algebraic specifications. The guiding thread of our approach is the Nereus Language, a Domain Specific Language for formal metamodeling. We show how to gradually introduce formal methods in the Software Engineering curriculum. Nereus can be linked to several demonstrators of theorems harnessing the full power of formal methods. A set of tools that support our approach is described.

Keywords: Software Education, Model Driven Engineering, Metamodeling, Software Engineering, Formal Methods, Algebraic Specifications, Theorem Provers.

1. INTRODUCTION

The teaching of specifications and formal methods plays a crucial role in the training of software engineering professionals. Formal methods provide systematic and rigorous techniques to reduce ambiguities and inconsistencies in software development. However, formal methods are controversial and, in general, they are only applied in the development of critical software. Their detractors think that software failures are an inevitable evil and do not encourage the use of formal techniques to prevent errors. Other authors argue that making software without failures is expensive. The challenges to adopting them in the industry are related to problems of understandability and scalability. Within the next few decades, tools based on verification will be as useful and widespread for software development as they are today in critical systems (Beckert & Hahnle, 2014). In many curricula, the specification and verification courses are not mandatory and have a marginal presence. The current situation in the software industry is that great advances in software technology go hand in hand with spectacular software failures which could be resolved by applying formal techniques.

Behind the specifications and formal methods exists a community that has developed theories, languages, methods, and tools, but the software industry does not take advantage of them, in particular, to effectively address the software crisis (Astesiano, Kreowski, & Krieg-Bruckner, 1999). There is already a substantial body of research, tools and case studies demonstrating that it is possible to develop software as reliable as any other engineering product. To enhance the software industry, artifact analysis tools that support novel combinations of code analysis techniques, model checking, testing and theorems provers are emerging (Beckert & Hahnle, 2014). In addition, formal methods can fill the requirements of new software development technologies. For instance, moving to the Cloud requires new ways to protect data and privacy. As well, the paradigm of IoT requires analyzing complex software/hardware systems in the early stages of design, mainly in software systems that will be embedded in systems deployed in different places. In both situations, formal methods can help to solve these problems.

Software failures can be detected in two major phases of the development process: specification or modeling, and programming. Taking into account that current students will not generate code in the conventional programming style we should adjust the verification and validation activities to new approaches (Cowling, 2015). In the future, Model Driven Engineering (MDE) becomes more widely applied (Brambilla, Cabot & Wimmer, 2017). MDE is a software development methodology that focuses on the use of models and model transformations to raise the level of abstraction and automation in software development, either to generate new software or to modernize legacy software. Model-driven principles can be summarized as follows: all artifacts involved in an MDE process can be viewed as models that conform to a particular metamodel, the process itself can be viewed as a sequence of model transformations and, all extracted information is represented in a standard way through metamodels. Model Driven Development (MDD) refers to forward engineering processes that use models as primary development artifacts. A specific realization of MDD is the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG) in the context of object-oriented modeling (MDA, 2014) (OMG, 2018).

Metamodeling is crucial in Model-Driven processes and one of the most important changes in teaching Software Engineering will result in the need to move the focus from models to metamodels that describe the structure of a family of models and the transforming methods at metamodel level for mapping them into code. The testing activities need to change from code-centric concepts to metamodel-based ones. It is important to reason about metamodels properly due to having errors in a metamodel leads to having errors in its model instances.

The main obstacle so that MDE impacts on software development is the lack of human resources that, through adequate training, assess the power of them and transfer their experience to software development projects. The construction of metamodels is not well supported with established practices and methodologies.

One of the most challenging aspects of teaching formal methods in undergraduate computer programs is that of finding the best approach to introducing the subject (Spichkova & Zamansky, 2016). It is essential to gradually introduce students to formal methods from the early stages of teaching Software Engineering (Ishikawa, Yoshioka & Tanabe, 2015). In this paper we propose to teach concepts of Model Driven Engineering and, metamodeling by integrating semiformal specifications with algebraic specifications. The bases of this approach are introduced in the early stages of teaching. The guiding thread of our approach is the Nereus Language, a Domain Specific Language (DSL) for formal metamodeling (Favre, 2009) (Favre & Duarte, 2016). We show how to gradually introduce formal methods going from the teaching of classes, relationships and object-oriented models in elementary courses of Algorithms to the teaching of formal metamodeling in advanced courses of Model Driven Engineering. Nereus can be linked to the Common Algebraic Specification Language (CASL) and through it, to several demonstrators of theorems harnessing the full power of formal methods. The main contributions of this work are: the Nereus language, a set of educational tools and a methodology to introduce formal specification in Software Engineering curriculum.

The structure of the article is as follows. Section 2 provides definitions and discussion on MDE. Section 3 introduces metamodeling concepts. Section 4 describes the syntax of the Nereus Language. In Section 5 a set of tools to make formal metamodeling feasible in practice is presented. Section 6 describes how to introduce algebraic specifications in undergraduate courses in a Software Engineering career and presents motivation remarking our contribution. Finally, in Section 7 we present conclusions.

2. MODEL DRIVEN ENGINEERING

Model Driven Engineering is a software development methodology that focuses on the use of models and model transformations to raise the level of abstraction and automation in software development, either to generate new software or to modernize legacy software. Different acronyms are associated with model-driven developments: MBE (Model Based Engineering), MDE (Model Driven Engineering), MDD (Model Driven Development), MDA (Model-Driven Architecture), MDSM (Model Driven Software Modernization) and ADM (Architecture Driven Modernization). Figure 1 shows the relation between the different acronyms.

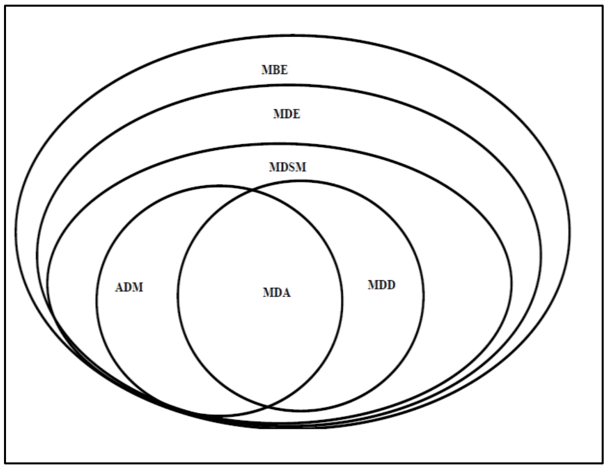


Figure 1. Model-driven Acronyms

MBE is the branch of software engineering in which software models play an important role, being the basis of development. However, there is no direct link between the models and the generated software precisely defined through transformations.

MDE can be viewed as a subset of MBE. It is the branch of software engineering in which processes are driven by models, i.e. models are the primary artifacts of different software processes. MDE has emerged as a new software engineering discipline which emphasizes the use of models and model transformations to raise the abstraction level and the degree of automation in software development. Productivity and some aspects of the software quality such as maintainability or interoperability are goals of MDE.

MDD refers to forward engineering processes that use models as primary development artifacts. A specific realization of MDD is MDA (MDA, 2014). The outstanding ideas behind MDA are, separating the specification of the system functionality from its implementation on specific platforms, managing the software evolution from abstract models to implementations. Models play a major role in MDA, which distinguishes at least platform-independent and platform-specific models. An MDA process focuses on the automatic transformation of different models that conform to MOF (Meta Object Facility) metamodel, the standard for defining metamodels in the context of MDA (MOF, 2016). It provides the ability to design and integrate semantically different languages such as general-purpose languages, DSLs and modeling languages in a unified way. MOF can be considered the essence of MDA allowing different kinds of artifacts from multiple technologies to be used together in an interoperable way. MOF provides two metamodels EMOF (Essential MOF) and CMOF (Complete MOF). EMOF favors the simplicity of implementation over expressiveness. The Eclipse Modeling Framework (EMF) was created for facilitating system modeling, metamodeling, and code generation (EMF, 2018). EMF started as an implementation of MOF resulting Ecore, the EMF metamodel comparable to EMOF. A variety of tools related to MDE are provided by EMF.

MDSM is a particular form of reengineering for the technological and functional evolution of legacy systems that begins to be identified in the early 21st century (Brambilla et al., 2017). It is based on model-driven processes of reverse engineering, restructuring and forward engineering. In the context of ADM, a set of modernization specifications is developed. ADM is defined as "the process of understanding and evolving existing software assets for the purpose of software improvement, modifications, interoperability, refactoring, restructuring, reuse, porting, migration, translation, integration, and service-oriented architecture deployment" (ADM, 2018).

3. FORMAL METAMODELING

MDE will impact on software engineering curriculum, in both the body of knowledge and the structure of its courses. In light of this, verification and validation will also change significantly and formal metamodeling will have to be addressed from different views.

The essence of MDA is the metamodel MOF allowing interoperability from multiple technologies. 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.

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). MOF metamodels are specified by using restricted UML (Unified Modeling Language) class diagrams and annotations OCL (UML, 2017) (OCL, 2014). 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. A major obstacle for specifying metamodels is that they must master two different languages, UML for capturing the domain structure and OCL for the definition of well-formedness rules. There are no guidelines to assist the metamodel construction through both paradigms. Cadavid, Combemale, and Baudry (2015) observe that all metamodels tend to have a small subset of concepts that are constrained by the OCL rules, most of them are loosely coupled to the underlying structure.

Our main contribution is the integration of MOF metalanguage with formal specification languages. In this context, we consider that 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 the 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. The inference system can help to automate testing, prototyping or verification.

Following the previous considerations, we define Nereus, a formal metamodeling language, and processes for reasoning about MOF-like metamodels such as Ecore metamodels.

It is important to remark that the instantiation of a metamodel produces models, which in turn are instantiated. So, 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. Current metamodeling tools enable code generation and detect invalid constraints, however, they do not find instances of the metalanguage (models). Formal methods offer rigor and precision while reducing ambiguity and inconsistency. Most MDE semiformal metamodels do not have support for typing metamodels and the notion of polymorphism at the metamodel level is imprecise due to the type of a metamodel is not defined. These are limitations for certain applications related to MDE. For instance, to have enough valid instances available is a requisite to test model transformations. Subtyping has consequences regards to reuse of models and metamodels and model transformations.

4. THE NEREUS LANGUAGE

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 Nereus syntax (classes, associations, and packages) and some examples of Nereus specifications. Next, we present some aspects of the semantic of Nereus.

Nereus Syntax

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 the 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 a 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 properties. OPERATIONS clause introduces the operation signatures, the list of their arguments and result types. Operations can be declared as total or partial. 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 functionality denoted by an underscore in the operation signature.

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) (Figure 2). The component Association provides Relation Schemes that can be used in the definition of concrete associations by instantiating classes,

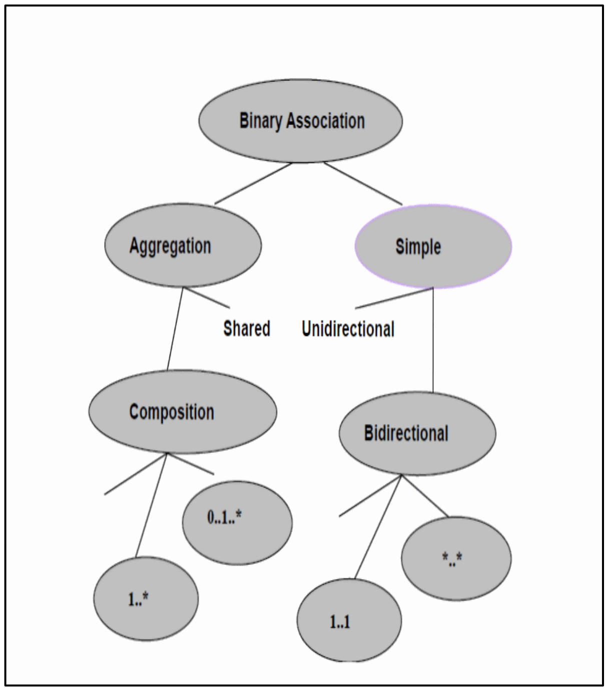


Figure 2. The Component Association

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>>

Defining Packages

The package is the mechanism provided by Nereus for grouping related model elements together in order to manage complexity and facilitate reuse. The package has the following Nereus syntax:

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

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.

Examples

Figure 3 shows the Nereus specification of the CLASS Heap. Figure 4 the specification of a fragment of the C++ metamodel. It shows three complementary specifications of a metamodel with which the student will work: an Ecore metamodel (Figure 4 a), its OCL restrictions (Figure 4 b) and the Nereus specification (Figure 4 c).

```
CLASS Heap
IMPORTS Nat
BASIC CONSTRUCTORS inicHeap, agregarHeap
EFFECTIVE
TYPE Heap
OPERATIONS
inicHeap: -> Heap;
agregarHeap: Heap * Nat -> Heap;
vacioHeap: Heap -> Boolean;
raizHeap: Heap(h) -> Nat
  pre: not vacioHeap(h);
eliminarHeap: Heap (h) -> Heap
  pre: not vacioHeap(h);

AXIOMS h:Heap; e: Nat;
vacioHeap(inicHeap()) = True;
vacioHeap(agregarHeap(h,e)) = False;
vacioHeap(h) => (raizHeap(agregarHeap(h,e)) = e);
(not vacioHeap(h) and e < raizHeap(h)) =>
raizHeap(agregarHeap(h,e)) = e;
 (not vacioHeap(h) and e >=  raizHeap(h))) =>
raizHeap(agregarHeap(h,e)) = raizHeap(h);
vacioHeap(h) => eliminarHeap(agregarHeap(h,e)) = inicHeap();
(not vacioHeap(h) and e < raizHeap(h)) =>
eliminarHeap(agregarHeap(h,e)) =  h;
(not vacioHeap(h) and e >= raizHeap(h)) =>
 eliminarHeap(agregarHeap(h,e))  = agregarHeap(eliminarHeap(h),e);
END-CLASS
```

Figure 3. The Class Heap

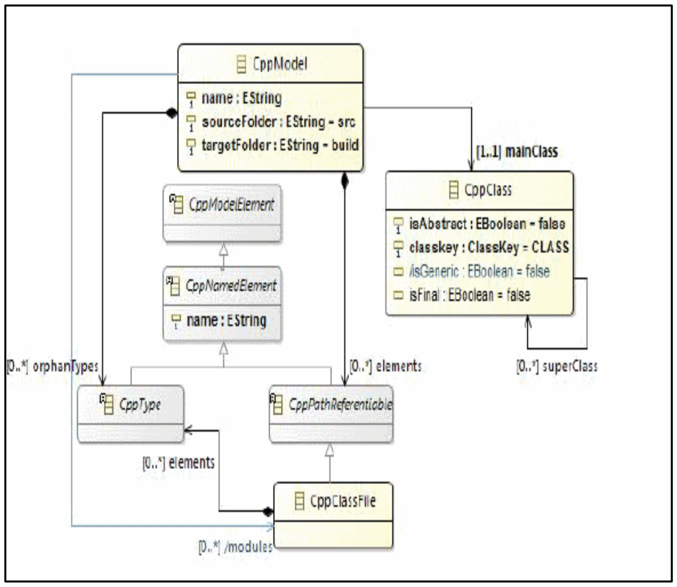


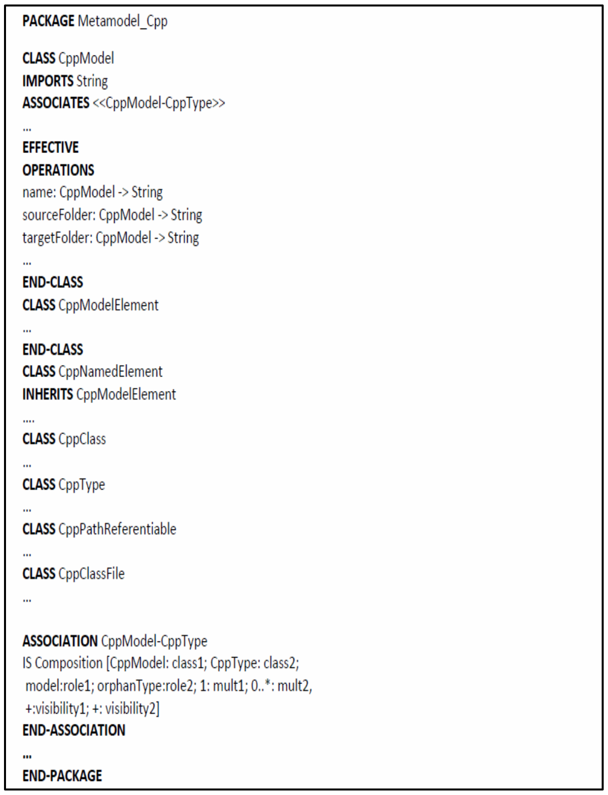
Figure 4 a. Fragment of the C++ Metamodel

```
package Metamodelo_Cpp : _'org.eclipse.emf.ecore.
metamodelo_cpp' = 'http://metamodelo_cpp.unicen.edu.ar'
{
  class CppModel
  {
    attribute name : String { id };
    attribute sourceFolder : String = 'src';
    attribute targetFolder : String = 'build';
    property mainClass : CppClass;
    property elements : CppPathReferentiable[*]
    { composes };
    property orphanTypes : CppType[*] { composes };
    property modules : CppClassFile[*] { derived
    readonly volatile }
    {
      derivation: elements
                  ->select(x | x.oclIsKindOf(CppPackage))
                  ->collect(x : CppPackage |
        x.childrenReferences )
                  ->select(y | y.oclIsKindOf(CppClassFile))
                  ->collect(y | y.oclAsType(CppClassFile));
    }
    invariant nameNotEmpty: name.size() > 0;
    invariant nameNotNull: name <> null;
  }
  abstract class CppPathReferentiable extends
  CppNamedElement
```

Figure 4 b. Fragment of the C++ Ecore Metamodel and OCL

Nereus Semantics

The semantics of Nereus was constructively given by translation to CASL (Bidoit & Mosses, 2004). 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



```
PACKAGE Metamodel_Cpp

CLASS CppModel
IMPORTS String
ASSOCIATES <<CppModel-CppType>>
...
EFFECTIVE
OPERATIONS
name: CppModel -> String
sourceFolder: CppModel -> String
targetFolder: CppModel -> String
...
END-CLASS
CLASS CppModelElement
...
END-CLASS
CLASS CppNamedElement
INHERITS CppModelElement
....
CLASS CppClass
...
CLASS CppType
...
CLASS CppPathReferentiable
...
CLASS CppClassFile
...

ASSOCIATION CppModel-CppType
IS Composition [CppModel: class1; CppType: class2;
 model:role1; orphanType:role2; 1: mult1; 0..*: mult2,
 +:visibility1; +: visibility2]
END-ASSOCIATION
...
END-PACKAGE
```

Figure 4 c. Fragment of the C++ Metamodel in Nereus

languages. It is supported by tools and facilitates interoperability of prototyping and verification tools. CASL is linked to ATP through HETS (Hets, 2018). We define a way to automatically translate each Nereus construct into CASL, including classes, different kinds of relations and packages. The most interesting problems in the translation are how to translate higher order functions, associations and packages. A detailed description may be found at (Favre, 2009).

5. EDUCATIONAL TOOLS

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). 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.
- 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 the process of the obtained structures. The translator from Nereus to CASL is based on the constructive semantics described previously in (Favre, 2009).
- 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.

Figure 5 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 5), 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. Different kinds of specifications could be selected (class, associations or packages).

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, and relation schemes) 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).

These tools are integrated with CASL and through it with Automatic Theorem Provers (ATP) provided by the Heterogeneous Tool SET (HETS) (Hets, 2018) (Mossakowski, Maeder & Codescu, 2014). HETS provides an open source general framework for formal method integration and proof management. Different logics and their analysis and proof tools can be used. ATPs allow performing a consistency analysis of the metamodel and achieving an analyzed specification by using a variety of theorem provers such as Isabelle and SPASS. HETS support a number of input languages such as CASL, OWL, Haskell, and Maude.

6. USING ALGEBRAIC SPECIFICATIONS

Formal methods are gradually introduced from the teaching of classes, relationships and object-oriented models in elementary courses of Algorithms to the teaching of formal metamodeling in advanced courses of Model Driven Engineering.

In a first level, students identify and specify algebraically the classes of objects that intervene in the problem. This specification describes the behavior of the classes in an abstract form, independently of particular implementations and allows performing early validations and verifying properties of the specification. The specifications of object classes are constructed from other existing ones with which it is related. Basic inheritance and client relationships are introduced in the specification of basic data structures such as stacks, queue, priority queues and collections (set, bag, ordered set or sequences).

Then, students tackle more complex problems that need to be modeled in more detail. In this stage, students work with semiformal specifications of UML class diagrams integrated with algebraic specifications in Nereus. Association relations, such as aggregation, composition, ordinary association, are introduced as first-class entities. From this specification, it is possible to perform validations that allow students to correct the specification that will be transformed into code. The level of specifications also facilitates to analyze possible implementations for both, object classes and relations in order to achieve an efficient implementation.

Finally, algebraic formalism is used in a course of Model Driven Engineering to specify MOF / Ecore metamodels. It is considered valuable to teach how to build metamodels taking into account that current designers will evolve into metadesigners. The objective of the course is to bring the students concepts of software modeling under the MDE approach, in particular following the principles and standards of MDA. To encourage students to learn formal specification, we must build-up their motivation by demonstrating the advantages of formal specifications in improving the production and reusability of high quality software. Upon completion of the course, the students will be able to build metamodels and models that conform to these metamodels, build model transformations and define MDE processes.

The process of meta-model construction is based on the analysis of fragments of models (examples of concrete instances) from which a metamodel is induced. These fragments focus on some interest aspects of the metamodel. In a first step, they are represented as an Ecore metamodel. A metaclass is created for each object of a different type (if it does not exist) and the attributes and operations and their restrictions are identified. Then, the different relationships between metaclasses are identified. The Ecore metamodel is subsequently transformed into a Nereus specification. 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 that, through its integration with CASL can be formally validated.

Students can experiment both by defining a metamodels for simple DSLs, and analyzing fragments of already defined metamodels, such as the Java metamodel, detecting inconsistencies and proposing solutions.

Figure 6 summarizes the typical flow with formal tools that is applied to analyze the specifications of classes, models or metamodels. First, a semiformal specification is transformed into a Nereus specification. 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 HETS. ATPs allow performing a consistency analysis of the metamodel and achieving an analyzed specification. The initial specification can be improved by reinjecting the changes introduced in the latter Nereus specification.

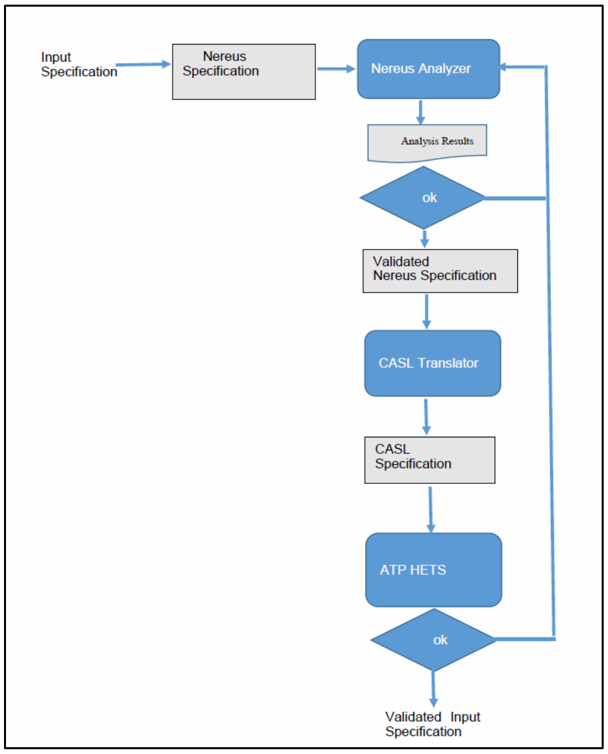


Figure 6. Using formal tools

The reasons for using Nereus as specification language are linked to pragmatic and educational 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 easily use metamodel constructs and 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 another 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. Another important issue is that Nereus, like MOF, provides the Package construct to structure large specifications in order to be legible and understandable. Also, it provides the additional benefit of supporting subtyping of packages.

7. CONCLUSION

One of the most challenging aspects of teaching formal methods in undergraduate computer programs is to find the best approach to present the topic. We believe that it is essential to gradually introduce Formal Method to students, beginning in the early stages of teaching. It is also convenient to show its benefits in terms of improving software engineering activities. Thus, we propose an integration of formal methods with model-driven developments. Our experience showed that formal methods can be applied to a real problem in advanced courses of MDE, in particular, the verification of properties of MOF metamodels.

The guiding thread of this approach is Nereus, a formal metamodeling language. Its syntax close to semiformal models in UML allows a harmonious integration in a curriculum that is based on the object-oriented paradigm both in the area of programming and in the area of software development methodologies. Key concepts such as classes, class diagrams (including relationships) and metamodels can be specified in a common framework supported by the Nereus language and through it with CASL, which in turn is integrated with demonstrators of theorems such as Isabelle and SPASS.

For several years, algebraic specifications have been used to introduce concepts associated with abstract data types (domain and operations) as we propose for a first undergraduate course. The original idea of this approach is to show how formal methods can be applied in real MDE projects to specify metamodels. This work showed only how the teaching of algebraic specifications can be integrated at different levels of education. There is a variety of formal methods. In (Beckert & Hahnle, 2014), 27 systems used in the verification of programs as a selection of the most representative of the state of the art, are analyzed. In different stages of student training and associated to different subjects, other formal methods could be taught based on a variety of different types of models, for example, the use of data models based on sets, relationships and functions, descriptions of interactions between processes through process algebra or the description of behavior processes in terms of machine models.

8. REFERENCES

ADM (2018). *Architecture-driven modernization task force*. Retrieved September 5, 2018 from http://www.adm.org.

Astesiano, E., Kreowski, H., & Krieg-Bruckner, B. (Eds.). (1999). Algebraic Foundations of System Specification. Heidelberg: Springer-Verlag.

Beckert, B., & Hahnle, R. (2014). Reasoning and Verification: State of the Art and Current Trends. *IEEE Intelligent Systems*. January/February 2014, 20-29.

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.

Brambilla, M., Cabot, J., & Wimmer, M. (2017). Model-Driven Software Enginneering in Practice, Second Edition, Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Cadavid, J.J., Combemale, B., & Baudry, B. (2015). An analysis of metamodeling practices for MOF and OCL. *Computer Languages, Systems and Structures*. 41, (2015), 42-65.

Cowling, A. J. (2015). The Role of Modelling in Teaching Formal Methods for Software Engineering. *Proceedings of the First Workshop on Formal Methods in Software Engineering Education and Training***.** Oslo, Norway. Ceur Worshop Proceedings, Vol 1385.

EMF (2018). *Eclipse Modeling Framework*, Retrieved September 5, 2018 from www.eclipse.org

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

Favre, L., & Duarte, D. (2016). Formal MOF Metamodeling and Tool Support. *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. MODELSWARD 2016,* 99-110.

Hets (2018). *Heterogeneous Tool Set*. Retrieved September, 2018 from http://www.informatik.unibremen.de/agbkb/forschung/formal_methods/CoFI/hets/

Ishikawa, F., Yoshioka, N., & Tanabe, Y. (2015) Keys and Roles of Formal Methods Education for Industry: 10 Year Experience with Top SE Program. *FMSEE&T@FM 2015*, 35-42.

MDA (2014). *Object Management Group Model-driven Architecture (MDA) MDA Guide rev. 2.0.* Retrieved September, 2018 from OMG Document ormsc/2014-06-01

MOF (2016). *Meta Object Facility (MOF) Core Specification*, Version 2.5. OMG Document Number: formal/2016-11-01. Retrieved September, 2018 from http://www.omg.org/spec/MOF /2.5.1/

MOF (2006). OMG Meta Object Facility (MOF) Core Specification, version 1.0. Retrieved September, 2018 from http://www.omg.org/spec/MOF

Mossakowski, T., Maeder, C., & Codescu, M. (2014). *Hets User Guide, version 0.99*. Retrieved September, 2018 from http://www.informatik.unibremen.de/agbkb/forschung/formal_methods/CoFI/hets/

OCL (2014). *Omg Object Constraint Language (OCL), version 2.4*, formal/2014-02-03. Retrieved September, 2018 from www.omg.org/ocl/2.4

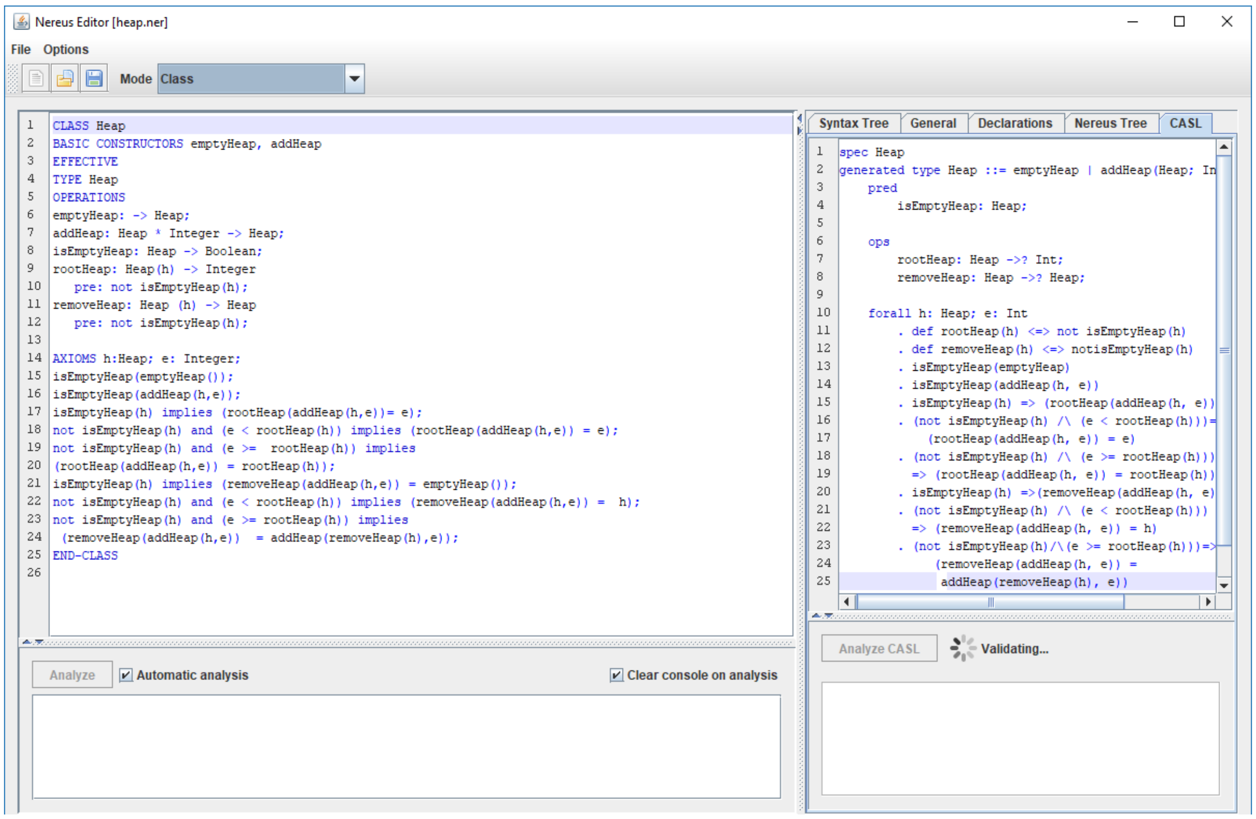
OMG (2018) *Object Management Group* Retrieved, September 2015 from www.omg.org

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

Spichkova, M., & Zamansky, A. (2016) Teaching of Formal Methods for Software Engineering *ENASE 2016 Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering, SCITEPRESS*, 370-377.

UML (2017) *Unified Modeling Language version 2.5.1.* Retrieved September 2018 from https://www.omg.org/spec/UML/2.5.1/

Appendices and Annexures



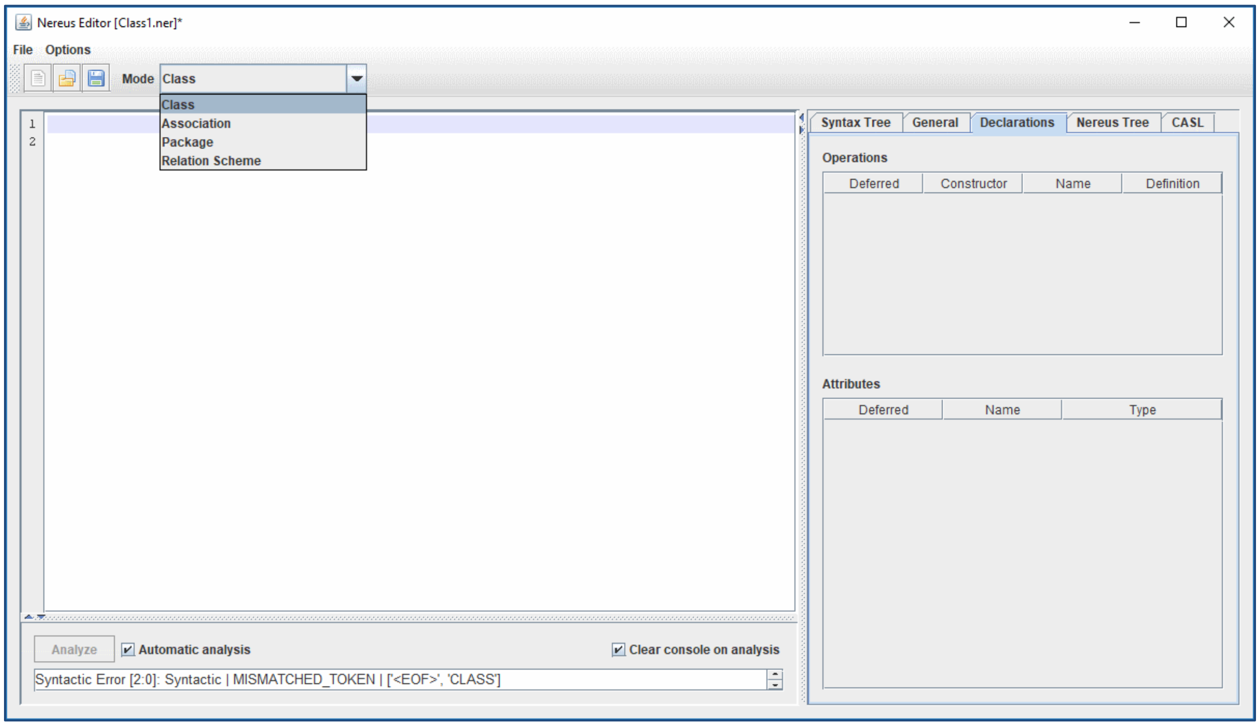
```
CLASS Heap
BASIC CONSTRUCTORS emptyHeap, addHeap
EFFECTIVE
TYPE Heap
OPERATIONS
emptyHeap: -> Heap;
addHeap: Heap * Integer -> Heap;
isEmptyHeap: Heap -> Boolean;
rootHeap: Heap(h) -> Integer
   pre: not isEmptyHeap(h);
removeHeap: Heap (h) -> Heap
   pre: not isEmptyHeap(h);

AXIOMS h:Heap; e: Integer;
isEmptyHeap(emptyHeap());
isEmptyHeap(addHeap(h,e));
isEmptyHeap(h) implies (rootHeap(addHeap(h,e))= e);
not isEmptyHeap(h) and (e < rootHeap(h)) implies (rootHeap(addHeap(h,e)) = e);
not isEmptyHeap(h) and (e >=  rootHeap(h)) implies
(rootHeap(addHeap(h,e)) = rootHeap(h));
isEmptyHeap(h) implies (removeHeap(addHeap(h,e)) = emptyHeap());
not isEmptyHeap(h) and (e < rootHeap(h)) implies (removeHeap(addHeap(h,e)) =  h);
not isEmptyHeap(h) and (e >= rootHeap(h)) implies
 (removeHeap(addHeap(h,e))  = addHeap(removeHeap(h),e));
END-CLASS
```

```
spec Heap
generated type Heap ::= emptyHeap | addHeap(Heap; In
    pred
        isEmptyHeap: Heap;

    ops
        rootHeap: Heap ->? Int;
        removeHeap: Heap ->? Heap;

    forall h: Heap; e: Int
        . def rootHeap(h) <=> not isEmptyHeap(h)
        . def removeHeap(h) <=> notisEmptyHeap(h)
        . isEmptyHeap(emptyHeap)
        . isEmptyHeap(addHeap(h, e))
        . isEmptyHeap(h) => (rootHeap(addHeap(h, e))
        . (not isEmptyHeap(h) /\ (e < rootHeap(h)))=
            (rootHeap(addHeap(h, e)) = e)
        . (not isEmptyHeap(h) /\ (e >= rootHeap(h)))
          => (rootHeap(addHeap(h, e)) = rootHeap(h))
        . isEmptyHeap(h) =>(removeHeap(addHeap(h, e)
        . (not isEmptyHeap(h) /\ (e < rootHeap(h)))
          => (removeHeap(addHeap(h, e)) = h)
        . (not isEmptyHeap(h)/\(e >= rootHeap(h)))=>
             (removeHeap(addHeap(h, e)) =
              addHeap(removeHeap(h), e))
```


Nereus Editor [Class1.ner]*
File Options
Mode Class
Class
Association
Package
Relation Scheme
Syntax Tree General Declarations Nereus Tree CASL
Operations
Deferred Constructor Name Definition
Attributes
Deferred Name Type
Analyze
Automatic analysis
Clear console on analysis

```
Syntactic Error [2:0]: Syntactic | MISMATCHED_TOKEN | ['<EOF>', 'CLASS']
```

Figure 5. Nereus Analyzer