

Definition of Modeling vs. Programming Languages

Maged Elaasar

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, USA
elaasar@jpl.nasa.gov

Abstract. Modeling languages (like UML and SysML) are those used in model-based specification of software-intensive systems. Like programming languages, they are defined using their syntax and semantics. However, both kinds of languages are defined by different communities, and in response to different requirements, which makes their methodologies and tools different. In this paper, we highlight the main differences between the definition methodologies of modeling and programming languages. We also discuss the impact of these differences on language tool support. We illustrate our ideas using examples from known programming and modeling languages. We also present a case study, where we analyze the definition of a new modeling language called the Ontology Modeling Language (OML). We highlight the requirements that have driven OML definition and explain how they are different from those driving typical programming languages. Finally, we discuss how these differences are being abstracted away using new language definition tools.

Keywords: Modeling, Programming, Syntax, Semantics, API, Methodology.

1 Introduction

Model-driven engineering (MBE) is a methodology that focuses on creating and exploiting models in the engineering of software-intensive systems. A model, expressed in a modeling language, is typically used to capture, communicate, and analyze the design of a software system. In a variant of the methodology, called model-driven development (MDD), the model is then transformed using a code generator into code in some programming language.

Both modeling and programming languages are computer languages that are defined in terms of their syntax and semantics. The syntax specifies the abstractions that can be used to describe a system, whereas the semantics specifies the meanings assigned to these abstractions. Moreover, the syntax can be specified in two levels: abstract and concrete. The abstract syntax specifies the abstractions or building blocks of expressions in a language (e.g., classes, fields, methods in the Java language) independently of their representations. The concrete syntax specifies the representations (using textual or graphical notation) of those abstractions. Both syntaxes are often mappable to each other, although the typical direction and completeness of these mappings may vary between modeling and programming languages.

The syntax of a programming language like Java, Scala or C++ is typically specified using a context-free grammar expressed in a notation like Backus-Naur Form (BNF) [1]. A BNF grammar consists of a set of terminal symbols, non-terminal symbols, and production rules (in the form $\langle \text{non-terminal} \rangle ::= \langle \text{expression} \rangle$) that transform each non-terminal into a sequence of terminals and/or non-terminals. This specifies the concrete textual syntax of a programming language. Moreover, the abstract syntax is also (automatically) derivable from such grammar. It is represented as an abstract syntax tree (AST) that is made up of non-terminal nodes. However, such AST is context-free. Performing context analysis (gathering and checking semantics) on the AST is typically encoded manually in some imperative programming language.

On the other hand, the abstract syntax of a modeling language, like UML [2], SysML [3] or BPMN [4], is typically specified using a meta (higher-level) modeling language like Meta Object Facility (MOF) [5]. A MOF-based metamodel specifies the abstractions of a language as meta classes, that have properties, operations, relationships (e.g., generalizations, compositions, cross references) to other meta classes, and well-formedness rules, expressed in a rule language like the Object Constraint Language (OCL) [13]. Thus, unlike a BNF grammar, a MOF metamodel captures the semantic context of a modeling language (at least partially). Moreover, the concrete syntax of a modeling language is typically defined independently of the abstract syntax. There can be several concrete syntaxes for a language and each of them can be textual and/or graphical. While there are some standards that can be used to describe those concrete syntaxes, and their relationship to the abstract syntax, they are typically defined less formally using English prose. On the other hand, there exist some de facto frameworks that are widely used to specify those concrete syntaxes. (More on this in section 2.2.)

Furthermore, modeling languages differ from programming languages in terms of their tooling concerns and approaches. For example, modeling languages tend to have standard APIs, hence many general-purpose tools (e.g., query engines, transformation engines, visualizers) can be developed generically for them. Also, models are often stored in persistent storage in terms of their abstract syntax, as opposed (or in addition) to their concrete syntax. This persistence is often standardized (in XML or JSON). Also, models can grow in size dramatically and hence sometimes get persisted in databases (as opposed to files) to enhance their scalability. Also, in a collaborative editing environment, models often need to be compared and merged in terms of their AST as opposed to the persistent format. Models also often need to be visualized using a variety of notations and viewpoints.

In this paper, we show how the methodology of defining modeling languages often differ from that of programming languages, in terms of abstract syntax, concrete syntaxes and semantics. We also discuss the implications of these differences on language tooling concerns and approaches. We also report on a case study where a new modeling language, called the Ontology Modeling Language (OML) [6], has been defined. We highlight the requirements of OML, discuss how some of them may be different than typical ones for programming languages, and show how they have been addressed in the modeling language methodology. In addition, we reflect on the state of the practice in language definition today and highlight some technologies that have the potential to bring the two approaches closer together.

The rest of the paper is organized as follows. Section 2 describes the differences in the methodologies for defining programming vs. modeling languages. A discussion of the impact of these differences on language tooling is given in section 3. Section 4 presents a case study where the OML modeling language has been defined. Some reflections on the state of the practice are offered in section 5. Finally, section 6 concludes the paper and outlines future works.

2 Definition of Modeling vs. Programming Languages

In this section, we describe the different methodologies of defining programming and modeling languages. We structure the description along three dimensions: abstract syntax, concrete syntax, and semantics.

2.1 Programming Language Definition

Concrete Syntax. The concrete syntax of a programming language is typically textual and specified with a context-free grammar that is expressed using a common notation like Backus-Naur Form (BNF) or Extended BNF (EBNF) [7] (which helps deal with some limitations of BNF like the definition of repeatable elements). Such notation allows specifying the textual syntax in two levels: a lexical level and a grammar level. The lexical level is specified with regular expressions that determine how characters form tokens (terminals). The grammar level is specified with production rules that determine how tokens (terminals) form phrases (non-terminals). For example, the following BNF grammar snippet specifies the syntax of simple algebraic expressions:

```
<expr> ::= <term> "+" <expr> | <term>
<term> ::= <factor> "*" <term> | <factor>
<factor> ::= "(" <expr> ")" | <constant>
<constant> ::= number
```

With such grammar, the expression $(1 + 2) * (3 + 5)$ can be represented as a valid expression in the grammar. First, a lexer turns the sequence of characters into terminal tokens (e.g., $($, 1 , $+$, 2 , etc.). Then, a parser groups the terminals into non-terminals that are added as nodes in an abstract syntax tree (AST) using the BNF production rules. For example, the AST of the above expression can be represented in memory as $[\text{term}, [\text{factor}, [\text{expr}, \text{constant}, \text{constant}]], [\text{factor}, [\text{expr}, \text{constant}, \text{constant}]]$, where each node is represented as $[\text{parent}, \text{child1}, \text{child2}, \dots]$. When such AST is printed, it can show as $['*', ['(', ['+', '1', '2'], ')'], ['(', ['+', '3', '5'], ')']]$. Both the lexer and the parser for a programming language can usually be auto-generated from the BNF grammar of a language, as supported by tools like Lex-Yacc [8] or Antlr [9].

Abstract Syntax. As mentioned before, BNF is a context-free grammar, which means the AST produced based on it is also context free. This means that the AST just shows the composition of the non-terminals, without interpreting what they semantically mean

nor how they are related semantically to each other. This is left to an interpreter that processes the AST and either adds to it semantic context, or produces another tree with semantic context. To clarify what is meant by this, consider the following example program file (example.java) expressed in the Java language:

```
package a;
import a.b.C;
public class D extends C {
}
```

This file could be parsed using a (pseudo) Java BNF grammar as ['example.java', [package, 'a'], [import, 'a.b.C'], [class, 'D', [super, 'C']]]. With this AST, we can see the structure of the Java file, but what we cannot see yet is how the nodes of that structure relate to each other. For example, the fact that class D belongs to package 'a', and that class C belongs to package 'a.b' is not automatically inferred by the parser. Rather, this information is added by the Java compiler that processes the AST to add the type information and creates the cross references. Only then, the AST of a program becomes ready to be checked for well-formedness. A compiler is usually coded manually as a visitor pattern over the AST.

Semantics. The semantics of a programming language refers to the formal meanings of the abstractions of a language. Three kinds of semantics can be identified:

- *Denotational semantics:* where language abstractions are mapped to mathematical objects that describe the meanings of those abstractions.
- *Axiomatic semantics:* where assertions about language abstractions and their relationships to other abstractions are specified
- *Operational semantics:* where abstractions are interpreted as transitions between state in some state machine.

While some of these semantics (e.g., axiomatic) can be checked by a compiler, others (e.g., operational) are checked at run-time. Both the compiler and the runtime system are different applications that are written separately.

2.2 Modeling Language Definition

Abstract Syntax. The abstract syntax of a modeling language is explicitly specified as a metamodel (a higher-level model) in a formalism such as the Meta Object Facility (MOF), defined by the Object Management Group (OMG). MOF is used to define many popular modeling languages like UML and BPMN. MOF is defined in two levels, Complete MOF (CMOF) and Essential MOF (EMOF). The latter is much simpler and has more adoption in the industry thanks to its popular Java implementation on the Eclipse platform called Ecore, which is provided by the Eclipse Modeling Framework (EMF) [11]. Moreover, Ecore itself has a textual syntax called Xcore [12].

A metamodel, defined in Ecore, specifies a modeling language as a set of interrelated classifiers. A classifier can either be a primitive type or a class. Primitive types, like Integer, Boolean, and String classify literal values. Classes, on the other hand, represent abstractions in the language. A class can have a number of structural features and operations and can specialize a number of other classes (in a taxonomy). A structural feature can either be an attribute, typed by a primitive type, or a reference, typed by a class. An operation, which may have a number of input parameters and a return type, represents behavior offered by the class. A class can also be constrained by a set of invariants expressed in a language called OCL, which supports a subset of first order predicate logic. In addition to invariants, OCL can also be used to specify body conditions of operations (i.e., conditions on the results of operations).

Unlike the abstract syntax of programming languages, which is generated in memory as a result of parsing and not persisted, that of modeling languages is typically persisted independently of the concrete syntax, and especially when the concrete syntax is only partial, i.e. does not represent all the information. A common specification for persisting MOF models is called the XML Metadata Interchange (XMI) [14], which maps the MOF syntax to XML.

It is also worth noting that MOF is not the only formalism to specify modeling languages in. Another common one is a UML profile, which is UML's extensibility mechanism. A UML profile creates a language with abstractions (called stereotypes) that are extensions of some meta classes of UML. One of those profile-defined languages is SysML, which extends UML to allow for modeling systems. An example of a SysML stereotype is called Block, which extends Class from UML to model a physical block.

Concrete Syntax. The concrete syntax of programming languages is usually defined as a view on its abstract syntax. Specifically, for meta classes in a metamodel, there could be rules in a concrete syntax specification that define how instances of those meta classes are depicted. A modeling language can have one or more concrete syntaxes, each of which can be textual or graphical.

Textual Concrete Syntax. While there is no dedicated specification for the specification of the textual syntax of a modeling language, there is a specification from OMG, called Model to Text (MTL) [15,] that can be used to map the abstract syntax to some textual notation. The specification allows defining a textual template with embedded tags that have OCL expressions that query the model and convert the information into text. An implementation of this specification for EMF-based models exists and is called Aceleo [16]. However, MTL does not aim at producing a canonical mapping from an abstract syntax to its textual syntax, nor does it support the other mapping direction.

Furthermore, one technology that has become the de facto standard for defining the textual syntax of EMF modeling languages is called Xtext [17], which supports bidirectional mapping between an EMF-based metamodel and an EBNF grammar for the language. Xtext supports both the generation of a metamodel from a given EBNF grammar and vice versa. Either way, Xtext provides the ability to specify a model using text that conforms to an EBNF grammar and automatically parses it in memory into the corresponding instance of the metamodel. The opposite direction is also supported.

Graphical Concrete Syntax. Modeling language specifications have historically described the graphical syntax informally using English prose. However, recently, an OMG specification, called Diagram Definition (DD) [18], emerged to address this limitation. DD allows the specification of a graphical syntax by specifying a unidirectional mapping from the abstract syntax metamodel of a modeling language to a graphical notation metamodel (e.g., of 2D graphics). This model-to-model mapping can be specified using a MOF-based mapping language like the Query/View/Transformation (QVT) [19]. An implementation of this specification is provided for EMF in the Papyrus modeling tool [20].

Furthermore, there exist several technologies that support the implementation of graphical syntaxes for EMF-based modeling languages. The first is the Graphical Modeling Framework (GMF) [21] and second is Sirius [22]. Both technologies support bi-directional mapping between an abstract syntax model and its graphical notation. However, GMF supports an imperative-style mapping, while Sirius supports a declarative-style mapping (using a mapping model).

Semantics. Unlike a BNF grammar for a programming language, the abstract syntax for modeling language is not context-free, but rather declaratively specifies the axiomatic semantics of the language. For example, the UML metamodel specifies that meta class `UML::Class` can reference another `UML::Class` as its superclass, contains one or more `UML::Property` as attributes, each of which references its `UML::Type`, etc. Also, the `UML::Type` meta class has a well-formedness rule expressed in OCL as *'not self->closure(supertype)->contains(self)'*, which means a type cannot have cyclic inheritance. This means just by using the metamodel, we can check the conformance of a model to the abstract syntax, including its axiomatic semantics.

However, unlike programming languages, many modeling languages do not have (at least complete) denotational nor operational semantics. For example, the semantics of the UML has a fair amount of variability that is open to semantic interpretation. For example, the direction of the association arrows indicate that it is efficient to navigate from the source object to the target object of the association, but it is a variation point what this exactly means. The focus of those modeling languages is on communication of ideas rather than reasoning, simulation or execution. For example, the Business Motivation Model (BMM) [23] language is a way to describe the strategy for a business.

However, some modeling languages have formal semantics, and in those cases the (denotational and operational) semantics are specified similarly to programming languages. For example, Foundational UML (fUML) [24] is an executable language that provides well-defined operational (execution) semantics for a subset of UML.

3 Tooling of Modeling vs Programming Languages

In this section, we discuss the differences between the tooling concerns of modeling and programming languages. These differences are mostly due to differences in language engineering requirements between the two. We structure the discussion along several axes, and relate them to the methodologies in section 2 then it makes sense.

3.1 API

The API of a language refers to the interface through which a description specified in the language can be manipulated by tools on a given platform. Examples of such tools are editors, compilers, visualizers, interpreters, etc. In the case of a programming language, there is typically no specification of an API, at least not a single API. However, tools, like Antlr, can usually generate an API that corresponds to the context-free AST of the language, along with a visitor pattern that can be implemented to add semantic annotations (e.g., typing and cross references) or generate another context-specific API, which can be used to perform further analysis like execution, visualization, etc.

On the other hand, modeling languages tend to have an API that is fully derived from the metamodel, hence already incorporates all typing and cross referencing. Such API includes a factory pattern to instantiate every concrete class, getter and setter methods to access properties of classes and methods corresponding to the class operations. In addition, the generated API often extends an abstract (reflective) API provided by the tool (e.g., EMF) for all languages. Such API allows the definition of generic tools (e.g., editors, query engines, transformation engines) that work on different modeling languages in a consistent fashion.

3.2 Editors

An editor for a language allows creating descriptions in that language. They support features like content assist, auto completion, validation, syntax highlighting, formatting, refactoring, etc. Many of these features have some aspects that apply generically to all languages (in a given family), and some aspects that are specific to a language. Therefore, it is common to implement editors using frameworks that provide those generic aspects and can be extended to support the specific of a given language.

One such framework for programming languages is the Language Server Protocol (LSP) [25], which defines a protocol used between an editor and a language server that provides services like auto completion, go to definition, find all references, etc. The LSP has been implemented by several editors like Eclipse and Visual Studio Code for a variety of programming languages like Java, Python, and C++. In addition, editors for modeling languages with textual syntaxes, like those defined with Xtext, also implement the LSP. But, thanks to the reflective API provided by EMF, the implementation of LSP can be done in a generic way for all modeling languages.

Similarly, modeling languages with graphical syntaxes have editor frameworks (e.g., GMF and Sirius) that can support modeling languages generically thanks to the reflective API of EMF. For example, deleting an object from a model can almost always be implemented generically for any language, since removing an object from a model removes all its contained elements and their cross references.

It is also worth noting that since a modeling language can have multiple (textual or graphical) concrete syntaxes, that their editors or IDEs can allow working with those different concrete syntaxes or switching between them.

3.3 Persistence

Both programs and models can be stored in persistent storage. However, they differ in what gets stored and how it is stored. For a programming language, the concrete textual syntax is what gets stored and it is almost always stored in files. The abstract syntax, on the other hand, is derived in memory when the files are parsed by compilers.

On the other hand, for a model, what gets stored can be either the abstract syntax only, the concrete syntax only, or both. For example, when a language has a textual concrete syntax, that syntax is what gets stored only, since the abstract syntax can unambiguously be derived upon parsing. However, when a language has a graphical syntax, both the graphical syntax and the abstract syntax are stored (together or in separate storage). In this case, the graphical syntax would contain its own details (layouts, styles) and reference the abstract syntax for the semantic details. Moreover, unlike a program, a model can also be defined exclusively using its abstract syntax APIs. In this case, only the abstract syntax is stored. When the model is stored in files, there are standards (e.g., XMI) that provide generic persistence rules based on the model's metamodel. This relieves the language developer from specifying a language-specific persistence format.

Furthermore, unlike a program, a model can also be stored in a database. While there are no standards for this, there are some technologies that support this like Connected Data Objects (CDO) [26], which is able to persist an EMF model in databases. This is possible thanks to the metamodel of the language and its EMF reflective API.

3.4 Configuration Management

Configuration Management (CM) refers to the ability to manage a program or a model in a repository that supports branching and version control. Many file-based CM systems, like Git [27], supports this generically. An important feature of those systems is the ability to compare versions of the files and calculate their differences. This supports auditing the change history of a file, but is also a prerequisite for merging changes to the same file by different contributors, which may have conflicts to resolve.

The compare/merge feature is therefore important to both programs and models since they are expected to have multiple contributors. In the case of programs and models with textual syntaxes, the typical support calculates textual differences. However, such support is not ideal because non-conflicting changes may sometimes appear on the same lines, hence often get misclassified as conflicts.

In the case of models that have abstract syntaxes, textual differencing is not effective as users do not typically work with the textual storage format directly. Instead, thanks to the reflective API of metamodels, there exist differencing and merging frameworks (e.g., EMF Compare [28]) that work on the model level, showing the users the changes in terms of the abstractions they work with. However, those changes are often at a lower level of abstraction than the actual editing operations that caused them in the editor. That is why some differencing frameworks offer ways to aggregate the lower changes into higher level changes. An example of this situation is when a model also has a graphical syntax that is stored separately. Changes in related abstract and graphical syntaxes can be correlated and presented together.

3.5 Extensibility

Extensibility refers to the ability of syntax or semantics of a language to be extended to cover other concerns. In the case of programming language, this is often a feature that needs to be built into the language. For example, Java supports the notion of annotations that can be put on its main abstractions (e.g., classes, attributes, methods) to add more information to them. In this case, interpreters can be developed to process those annotations (e.g., insert extra code in methods to implement a `@precondition` annotation).

Similarly, modeling languages can each have its own extensibility mechanism. For example, UML's extensibility mechanism is the UML profile, which provides a set of stereotypes that can apply to UML elements to tag them or possibly add new attributes. However, another extensibility mechanism may be defined by the meta language itself, thus inherited by all its languages. For example, EMF provides an abstract class that supports having key-value annotations. Meta classes inheriting from this abstract class give themselves this extensibility feature.

3.6 Integration between Languages

It may sometimes be desirable to integrate different languages together. For example, one programming language may embed expressions from another programming language that is better able to address a concern (e.g., it is possible to embed expressions in Assembly within a C program). Another example is a modeling language that allows its elements to reference elements from another language (e.g., BPMN allows its elements to reference elements from any other model including UML). In both cases, the language definition can be designed to allow such support (both BNF and MOF allow a language to import definitions from other languages).

However, in modeling languages, this integration can also be free if one language is an extension of another (or of a common parent language). This is because when a class in a modeling language references another class, objects of the first class can reference objects of the second class or any of its subclasses, even if they are not defined by the same language. Moreover, if the reference is typed by the most abstract base class (defined by the meta language), then it can point to any object in any language.

4 Case Study

In this section, we report on a case study where we analyze the methodology of defining a new modeling language for a data warehouse system. We show how the language engineering requirements for this modeling language differ from those of a typical programming language, which also explains the differences in methodology and tool support. We first define the language requirements then analyze them. After that, we define the new language itself, specifically its abstract syntax, concrete syntax and semantics. Finally, we explain how this definition addresses the language requirements.

4.1 Language Requirements

The language we focus on is called the Ontology Modeling Language (OML), which was developed at the Jet Propulsion Laboratory (JPL) to be the lingua franca of a new data warehouse system for systems engineering. The basic requirements for the language include the ability to represent multi-viewpoint descriptions of a system that are authored in different system authoring tools, check the well-formedness and consistency of those descriptions, configuration manage them, and make them available for analysis and reporting.

4.2 Requirement Analysis

The main requirement of the language is support multi-view knowledge representation. Hence, the first candidate language that was considered for that was the Web Ontology Language (OWL) 2 [29]. OWL 2 has desirable characteristics that can help address this problem: a) it supports organizing information as triples (subject, predicate, object) in different ontologies, which facilitates tracking the provenance of information coming from each authoring tool, and b) supports storing the triples in textual files using multiple formats (e.g., XML, JSON, N-triples), which can be easily configuration managed in widely available file-based C/M systems (like Git).

Moreover, since OWL 2 supports multiple sublanguages, one of them was chosen, which was OWL 2 DL (Description Logic). This sublanguage a) has description logic semantics, which provides good expressive power for system descriptions, b) supports inference-based reasoning, which can help check the well-formedness and consistency of system descriptions, c) has a defacto Java based API called Jena [30], and d) supports an expressive pattern-based query language called SPARQL [31].

However, OWL 2 DL also has some undesirable characteristics for use on this problem, including: a) there are multiple ways of using OWL 2 DL to encode the system descriptions, some of which can lead to unexpected results or cause scalability issues during the reasoning process, b) the OWL syntax is at a lower level of abstraction making the system descriptions verbose, c) the storage of triples in a file is not stable as the collection is not ordered, which may lead to unexpected deltas when ontologies (generated automatically from authoring tools) get committed to a C/M system, d) the language has no visual notation that can help abstract out the information for users, e) the Java API is mutable, which makes it harder to use to perform expensive analyses in a distributed computing environment (e.g., Spark [32]), and f) the Java API cannot be used on an important tooling platform, which is a web browser.

4.3 OML Language Definition

The list of desirable and undesirable characteristics above has motivated the design of a new language, called OML, that retains the benefits of OWL 2 DL but also addresses its limitations in this context. In this section, we discuss the design of OML, specifically in terms of its abstract syntax, concrete syntax, semantics and API.

Abstract Syntax. OML is designed as a modeling language for ontologies, hence follows the common practice of defining the abstract syntax first with a metamodel. In this case, we chose to define it in Ecore (to leverage EMF’s large ecosystem of tools), and in particular using its Xcore textual syntax. A simplified subset of the language metamodel in Xcore is shown in Fig.1 (for brevity).

```

package oml {
  class Terminology {
    String iri
    Boolean isOpen
    contains Terms [*] terms
  }
  abstract class Term {
    String name
  }
  abstract class Entity extends Term {}
  class Aspect extends Entity {}
  class Concept extends Entity {}
  class Relation extends Term {
    refers Entity [1] source
    refers Entity [1] target
  }
}

```

Fig. 1. A simplified subset of the OML metamodel expressed in Xcore

The main abstractions (e.g., oml:Terminology, oml:Concept, oml:Aspect, oml:Relationship) are modeled as first class concepts in OML, as opposed to the much generic equivalent concepts in OWL 2 DL (e.g., owl:Ontology is used to represent either a terminology ontology or a description ontology, while owl:Class is used to represent a concept, an aspect, or a relationship between them). This makes modeling in OML more precise and also concise (more on this below).

Also, other desirable features of OWL 2 DL are maintained in OML. For example, the ability to preserve the provenance of information by keeping the information exported from every system tool in a separate OML terminologies is supported. Also, the ability to specify design patterns [10] in foundational (base) terminologies then specialize them in every discipline terminology is preserved. Moreover, the ability to create terminologies with open or closed world assumptions is retained in OML. Also, all the structural feature collections (e.g., Terminology;terms) in the metamodel are defined as ordered so that the persistence of an abstract model is stable.

Concrete Syntax. Several concrete syntaxes have been defined for OML to address different use cases. One of them is a textual syntax that was defined using an Xtext grammar (Fig. 2). The grammar is designed to work with the OML metamodel and is meant to be enable users to author ontologies with a text editor. This is mostly used to design terminologies representing the vocabularies of systems engineering disciplines. Such ontologies tend to be small in size, and configuration managed as text files.

```

grammar OML with org.eclipse.xtext.common.Terminals

generate oml "http://OML"

Terminology:
  (isOpen?='open')? 'terminology' iri=ID
  '{'
  (statements+=TerminologyStatement)*
  '}' ;

TerminologyStatement:
  Entity | Relation ;

Entity:
  Aspect | Concept ;

Aspect:
  'aspect' name=ID ;

Concept:
  'concept' name=ID ;

Relation:
  'relation' name=ID '{'
  'source' '=' source=[Entity]
  'target' '=' target=[Entity]
  '}' ;

```

Fig. 2. A simplified subset of the OML textual syntax expressed in Xtext

An example of a model in OML representing a simple terminology of system structure is shown in Fig. 3. The terminology defines two concepts, a Block and an Interface and a relationship Exposes between them. An equivalent model defined in OWL 2 DL turtle format is shown in Fig. 4. Notice how the OML syntax is more readable, concise and accurate (all terms in the OWL 2 DL format are defined with owl:Class making it harder to understand the Exposes relationship for example).

```

open terminology structure {
  concept Block
  concept Interface
  relationship Exposes {
    source = Block
    target = Interface
  }
}

```

Fig. 3. A simple model in OML textual syntax showing a structure terminology

```

<structure> a owl:Ontology .
<#Block> a owl:Class .
<#Interface> a owl:Class .
<#Exposes> a owl:Class .
<#source>
  a rdfs:Property ;
  rdfs:domain <#Exposes> ;
  rdfs:range <#Block> .
<#target>
  a rdfs:Property ;
  rdfs:domain <#Exposes> ;
  rdfs:range <#Interface> .

```

Fig. 4. A simple model in OWL 2 DL syntax showing a structure terminology

Another concrete syntax that was designed for OML is the OML Zip representation. In this representation, elements of the same type in one OML ontology are represented together as a JSON array, ordered by ‘iri’, and stored in a JSON document (Fig. 5). Then all JSON documents that belong to one ontology are put together into a zip archive. This syntax is not meant for users to author in directly, but is rather constructed by API when system descriptions are read from system authoring tools and converted to OML in order to be managed by the data warehouse. This representation is efficient for this use case since terminologies in this case tend to be much larger in size than vocabulary ontologies, hence will have a smaller footprint in this compressed syntax. This format also allows writing an algorithm to detect differences between two OML models very efficient, as deltas can be reduced down to changed lines, as opposed to structural deltas that result from complex structural comparisons. Finally, this format allows every type to be stored as an array of objects of the same length and shape, which makes loading them into columnar databases efficient for analysis.

```

"terminologies": {[
  {"iri": "structure", "isOpen": true}
]}

"concepts": {[
  {"iri": "structure#Block", "name": "Block"},
  {"iri": "structure#Interface", "name": "Interface"}
]}

"relationships": {[
  {"iri": "structure#Exposes", "name": "Exposes",
   "source": "structure#Block",
   "target": "structure#Interface"}
]}

```

Fig. 5. A simple model in OML JSON Zip format showing a structure terminology

The last concrete syntax for OML is a high-level diagrammatic syntax that resembles a class diagram, and can be used to visualize the system description terminologies. This helps users understand the system design. Such diagrammatic syntax has been defined using the Sirius framework (Fig. 6). A terminology is shown with a box whose name appears in top left corner, a concept appears as a box within it with a centered text, and a relationship appears as an arrow from its source to its target with its name below.

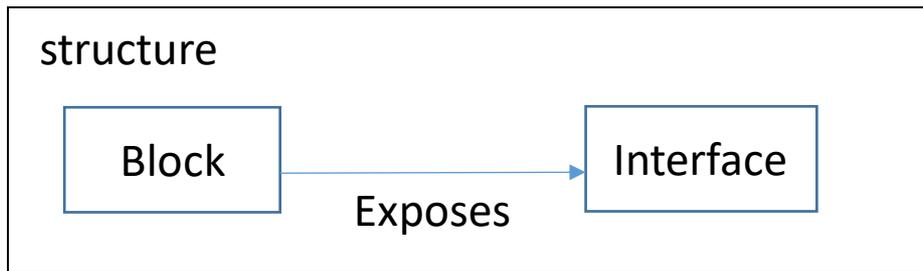


Fig. 6. A simple model in OML graphical notation showing a structure terminology

Semantics. The axiomatic semantics of OML has been encoded in the OML meta-model. However, its denotational semantics are defined by mapping its abstractions to those of OWL 2 DL, which has Description Logic semantics. This mapping (omitted here for brevity but alluded to by showing OML and OWL 2 examples in Fig. 3 and Fig. 4) is invoked before the OML ontologies is analyzed. The resulting OWL representation is loaded into a database (a triple store), then an inference engine is run on it to deduce new axioms from asserted axioms. Both sets of axioms can then be queried using a SPQRQL expression sent to a SPARQL query endpoint using the Jena API.

API. Defining the abstract syntax of OML with EMF provides it with a mutable Java API (e.g., factory pattern with named create methods, named getters/setter, etc.) that extends the framework-defined reflective API (e.g., EFactory::create, EObject::eGet, EObject::eSet, etc.). Such API allows OML to leverage the large EMF ecosystem of tools. For example, a textual editor was developed using the Xtext framework and a graphical editor was developed using the Sirius framework. Both editors can be used by users to author vocabulary terminologies. The Java API also simplifies the implementation of extract-transform-load (ETL) interfaces to system tools, which can read system descriptions, transform them to OML, and persist them in the OML Zip syntax.

In addition to the default Java API, two other APIs were generated for OML based on the same abstract syntax metamodel (using custom code generators). One of them is an immutable functional API in Scala. This API is used to read and query OML data for analysis purposes. Thanks to the immutability of this API, it makes writing analysis scripts with it safer and easier to distribute (for example, on a distributed computing platform like Spark). The last API is a JavaScript API that allows manipulating OML data in web browsers, which is used to visualize OML data in web applications.

5 Reflection

After having discussed how the requirements of modeling and programming languages can be different, hence can affect the language definition methodologies and tooling concerns, the real question is whether the two approaches are fundamentally different.

One may be tempted to think of modeling languages as typically informal, highly abstract, and visual until they learn that some modeling languages have very well-defined execution semantics and textual notation (e.g., ALF [33]). On the other hand, one may think of programming languages as fundamentally low-level and general-purpose, until they learn about high-level programming languages that translates into other languages (e.g., Xtend [34] translating into Java, and JSX [35] translating into JavaScript) and learn about domain-specific languages (e.g., R for statistical computations [36]).

It is important to realize that the language engineering requirements of both families of languages are getting closer and hence their methodologies and tools will too. For example, the Xtext framework allows a domain specific language to have both a meta-model and an EBNF grammar. Which class would you classify such a language in? It is in both. Another example is the LSP framework, which allows an editor to support multiple programming and/or modeling languages.

6 Conclusions and Future Works

Modeling and programming languages are both computer languages. However, there are different language engineering requirements deriving both of them. These usually stem from the difference in priorities and objectives of the communities defining them. This explain why the methodologies and tools for these languages are different.

In this paper, we highlight and discuss the different methodologies of defining modeling vs. programming languages, especially in terms of their abstract syntax, concrete syntax and semantics. We also discussed the implications of these differences on the language tooling, in terms of API, editor, persistence, configuration management, extensibility and integration between languages. We then reported on a case study where we analyzed how a new language, called OML, was designed to satisfy its unique requirements. We then reflected on how these design decisions have addressed the requirements and influenced the language's tool support.

Going forward, we plan to investigate ways to bridge the gap between modeling and programming languages, especially in terms of tool support. We believe that each community can learn a lot from the other. For example, we think that programming language tools should have structural compare/merge support that is akin to that available to modeling languages. We also think that the semantic of modeling languages should be formalized in similar ways to programming languages.

Acknowledgement

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Disclaimer

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Labor.

References

1. Hopcroft, J.; Ullman, J.: "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley, 1979.
2. Object Management Group: "OMG Unified Modeling Language, version 2.5.1," <http://www.omg.org/spec/UML/2.5.1/>, 2017.
3. Object Management Group: "OMG System Modeling Language, version 1.5," <http://www.omg.org/spec/SysML/1.5/>, 2017.
4. Object Management Group: "Business Process Model And Notation, version 2.0.2," <http://www.omg.org/spec/BPMN/2.0.2/>, 2014.
5. Object Management Group: "Meta Object Facility, version 2.5.1," <http://www.omg.org/spec/MOF/2.5.1/>, 2016.
6. Jet Propulsion Laboratory: "Ontology Modeling Language (OML) Workbench". <https://github.com/JPL-IMCE/gov.nasa.jpl.imce.oml.core>, 2018.
7. Scowen, R.: "Extended BNF — A generic base standard," Software Engineering Standards Symposium, 1993.
8. Levine, J., Mason, T., Brown, D.: "Lex & Yacc," O'Reilly & Associates, October 1992. ISBN: 1565920007
9. Parr, T.: "ANTLR," <http://www.antlr.org/>
10. Gamma, E., Helm, R., Johnson R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
11. Steinberg, D., Budinsky, F., Paternostro M., Merks, E.: "EMF: Eclipse Modeling Framework", 2nd edition, 2009.
12. Merks, E.: "Xcore", <https://wiki.eclipse.org/Xcore>
13. Object Management Group: "Object Constraint Language, version 2.4," <http://www.omg.org/spec/OCL/2.4/>, 2014.
14. Object Management Group: "XML Metadata Interchange, version 2.5.1," <http://www.omg.org/spec/XMI/2.5.1/>, 2015.
15. Object Management Group: "MOF Model to Text Transformation Language, version 1.0," <http://www.omg.org/spec/MOFM2T/1.0/>, 2008.
16. Obeo: "Acceleo," <https://www.eclipse.org/acceleo/>
17. TypeFox: "Xtext," <https://www.eclipse.org/Xtext/>
18. Object Management Group: "Diagram Definition, version 1.1," <http://www.omg.org/spec/DD/1.1/>, 2015.

19. Object Management Group: “MOF Query/View/Transformation, version 1.3,” <http://www.omg.org/spec/QVT/1.3/>, 2016.
20. CEA-LIST: “Papyrus,” <https://www.eclipse.org/papyrus/>
21. Eclipse: “Graphical Modeling Project (GMP),” <http://www.eclipse.org/modeling/gmp/>
22. Obeo: “Sirius,” <https://www.eclipse.org/sirius/>
23. Object Management Group: “Business Motivation Model, version 1.3,” <http://www.omg.org/spec/BMM/1.3/>, 2015.
24. Object Management Group: “Semantics of a Foundational Subset for Executable UML Models, version 1.3,” <http://www.omg.org/spec/FUML/1.3/>, 2017.
25. Microsoft: “Language Server Protocol,” <https://microsoft.github.io/language-server-protocol/>
26. Eclipse: “CDO Model Repository,” <https://projects.eclipse.org/projects/modeling.emf.cdo>
27. Git: <https://git-scm.com/>
28. Eclipse: “EMF Compare,” <https://www.eclipse.org/emf/compare/>
29. W3C: “OWL 2 Web Ontology Language Primer (2nd Edition),” 2012. <https://www.w3.org/TR/owl2-primer/>
30. Apache: “Apache Jena,” <https://jena.apache.org/>
31. W3C: “SPARQL Query Language for RDF”, 2008 <https://www.w3.org/TR/rdf-sparql-query/>
32. Apache: “Apache Spark,” <https://spark.apache.org/>
33. Object Management Group: “Action Language for Foundational UML, version 1.1,” <http://www.omg.org/spec/ALF/1.1/>, 2017.
34. Eclipse: “Xtend,” <https://www.eclipse.org/xtend/>
35. React: “Introducing JSX,” <https://reactjs.org/docs/introducing-jsx.html>
36. R-Project “The R Project for Statistical Analysis,” <https://www.r-project.org/>