Conceptual Modeling Language Specification

Version 1.0 (Draft)

Quenio Cesar Machado dos Santos

Universidade Federal de Santa Catarina*

November 2017

* Initially developed as part of the author's Bachelor Technical Report in Computer Sciences

Contents

I Language and Compiler 1

1 The Language 2

2 The Compiler 3
2.1 The Frontend 4
2.2 The Backend 4

3 Specification and Notations 5

II Conceptual Modeling 7

4 Concepts 8
4.1 Example 8
4.2 Syntax 8
4.3 Constraints 11

5 Properties 12
5.1 Example 12
5.2 Syntax 13
5.3 Constraints 14

6 Attributes 15
6.1 Example 15
6.2 Syntax 16

7 Derived Attributes 17
7.1 Example 17
7.2 Syntax 18

8 Associations **19**
8.1 Example 20
8.2 Syntax 22
8.3 Constraints 22
8.4 Translation 23

9 Derived Associations **25**

10 Generalization / Specialization **26**
10.1 Example 26
10.2 Syntax 27
10.3 Constraints 27

11 Abstractions **30**
11.1 Example 30
11.2 Syntax 31
11.3 Constraints 31

III Expressions **34**

12 Expressions **35**
12.1 Examples 35
12.2 Syntax 36

13 Literal Expressions **39**
13.1 Examples 39
13.2 Syntax 40

14 Arithmetic Expressions **41**
14.1 Examples 41
14.2 Constraints 41

15 Logical Expressions **43**
15.1 Examples 43
15.2 Constraints 43

16 Relational Expressions **45**
16.1 Examples 45
16.2 Constraints 45

17 Referential Expressions **47**

17.1 Examples 47
17.2 Constraints 47

18 Conditional Expressions **48**
18.1 Examples 48
18.2 Syntax 49
18.3 Constraints 49

19 Type Checking **50**
19.1 Examples 50

20 Type Casting **51**
20.1 Examples 51

21 String Concatenation **52**
21.1 Constraints 52
21.2 Type Inference 52
21.3 Translation 52

22 Path Expressions **54**

23 Invocation Expressions **55**

24 Lambda Expressions **56**

25 Comprehension Expressions **57**

IV Functions **58**

26 Built-In Functions **59**

27 Template Functions **60**

28 Declared Functions **61**

29 Comprehension Functions **62**

V Type System **63**

30 Types **64**

31 Primitive Types **66**

31.1 Example 66
31.2 Syntax 68

32 Boolean Type **70**

33 Numeric Types **71**

34 Floating-Point Types **72**

35 String Type **73**

36 Reference Types **74**

37 Tuple Types **75**

38 Function Types **76**

VI Code Generation **77**

39 Templates **78**

40 Constructors **79**

41 Tasks **80**

42 Targets **81**

VII Organization and Sharing **82**

43 Modules **83**

44 Libraries **84**

VIII Appendices **85**

A CML Concrete Syntax (Grammar) **86**
A.1 ANTLR Grammar 87

B CML Abstract Syntax (Metamodel) **91**

C CML Abstract Syntax Tree (Instantiation) **92**

D CML Constraints (Validations) 93

E Language Specification Notation 95

Bibliography 96

Listings

4.1 Concept Examples 9
4.2 Concept Concrete Syntax 9
4.3 Concept AST Instantiation 10
4.4 Concept Constraints 11

5.1 Property Examples 12
5.2 Property Concrete Syntax 13
5.3 Property AST Instantiation 13
5.4 Property Constraints 14

6.1 Attribute Example 16

8.1 Association Example 21
8.2 Association Concrete Syntax 22
8.3 Association AST Instantiation 23
8.4 Association Constraints 24

11.1 Abstract Concept Example 32
11.2 Abstract Concept Constraints 33

12.1 Expression Example 36
12.2 Expression Concrete Syntax 37

Figures

2.1 An architectural overview of the CML compiler. 3

4.1 Concept Abstract Syntax 10

10.1 Generalization Examples 28
10.2 Generalization Constraints 29

12.1 Expression Abstract Syntax 38

31.1 Example of *Primitive Types* 67
31.2 Type Declaration Syntax 69
31.3 Type AST Instantiation 69

Tables

13.1 Literal Expressions for Primitive Types 39
13.2 Literal Expressions for Primitive Types 40

14.1 Examples using the Arithmetic Operators 41
14.2 Arithmetic Operators in Precedence Order 42

15.1 Logical Operators in Precedence Order 43
15.2 Logical Operators in Precedence Order 44

16.1 Relational Operators 45
16.2 Relational Operators 45

17.1 Referential Operators 47

18.1 Conditional Expressions 49
18.2 Syntax of Conditional Expressions 49

19.1 Examples of Type-Checking Expressions 50

20.1 Examples of Type-Casting Expressions 51

31.1 Core Primitive Types in CML. 68
31.2 Additional Primitive Types in CML. 69

Part I

Language and Compiler

One

The Language

This document specifies the *Conceptual Modeling Language*, or CML for short. CML enables the modeling of the information of software systems. It focuses on modeling the structural aspects of such systems, having less emphasis on the behavioral aspects. Using CML, it is possible to represent the information as understood by the system users, while disregarding its physical organization as implemented by target languages or technologies.

In this first part of the CML specification, the first chapter will provide an overview of the CML compiler's architecture, and the second chapter describes the organization and notation used in the remainder of this document. The second part describes that structural constructs of the language that enable conceptual modeling. The third part covers values and expressions. The fourth part focuses on the semantics of type checking. The fifth part describes code generation. The last part will cover organization and sharing of conceptual models.

Two

The Compiler

The CML compiler has as *input*, source files defined using its own conceptual language (as specified in this document), which provides an abstract syntax similar to (but less comprehensive than) a combination of UML [4] and OCL [3]; and, as *output*, any target languages based on extensible templates, which may be provided by the compiler's base libraries, by third-party libraries, or even by developers.

The CML compiler's overall architecture follows the standard compiler design literature [2]. An overview diagram of the architecture is shown in figure 2.1. The two main components of the compiler, and the

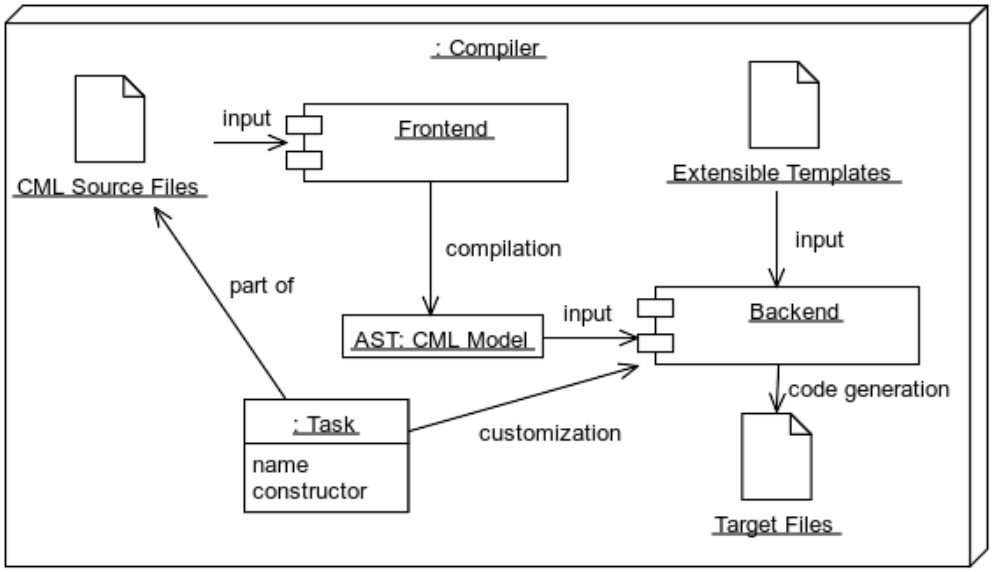

Figure 2.1: An architectural overview of the CML compiler.

artifacts they work with, are presented in the next sections.

2.1 The Compiler Frontend

The frontend receives as input the *CML source files*. It will parse the files and generate an internal representation of the *CML model*.

Syntactical and semantic validations will be performed at this point. Any syntax and constraint errors are presented to the developer, interrupting the progress to the next phase. If the *source files* are parsed and validated successfully, then the internal representation (the AST) of the *CML model* is provided as the input to the *backend* component.

2.2 The Compiler Backend

The backend receives the *CML model AST* as input. Based on the *target specification* provided by the AST, chooses which *extensible templates* to use for code generation. The *target files* are then generated, and become available to be consumed by other tools. The *task declaration* plays the key role of determining the kind of *target* to be generated.

CML extensible templates are implemented in StringTemplate [7]. The CML compiler uses StringTemplate for two purposes:

- *File names and directory structure:* each type of target generated by the CML compiler requires a different directory structure. The CML compiler expects each target type to define a template file named "files.stg" (also known as *files template*), which will contain the path of all files to be generated. The *files template* may use information provided by the *task declaration* (specified in chapter § 42) in order to determine the file/directory names.

- *File content generation:* each file listed under the *files template* will have a corresponding *content template* that specifies how the file's content must be generated. The *content template* will receive as input one root-level element of the CML model, which will provide information to generate the file's content. Each type of top-level model element should have a corresponding *content template*. Templates are described in *Code Generation* part of this specification.

Three

Specification and Notations

The first two parts of this specification specify all elements of CML metamodel needed for conceptual modeling. Every chapter in these two parts starts with a definition, which is followed by: an example; the specification of the concrete syntax; the abstract syntax; and then how to transform the concrete syntax into the abstract one.

The first paragrah of each chapter has an informal description of a CML metamodel element. If a correspondence exists to an element of the Entity-Relationship (ER) [1] metamodel, or to an element of the Unified Modeling Language (UML) [4] metamodel, it is provided with some comments of their differences.

Examples are provided in a separate section for each declaration of a metamodel element. These sections refer to a `verbatim` figure containing the examples, and describes them as needed. The examples are provided for illustrative purposes only, and they are *not* intended to be normative. They may be excerpts of larger CML source files, and thus may not be successfully compiled on their own.

The concrete syntax of each CML metamodel element is described on its own section. This type of section refers to a `verbatim` figure, which contains the actual ANTLR [6] grammar specifying the syntax for the CML metamodel element in question, and it must be considered normative. The appendix § A presents all the grammar rules in a single listing.

The abstract syntax of each CML metamodel element is described on the next section. This type of section refers to two types of figure: the first figure presents a class diagram with the EMOF [5]-based metamodel of the element being described; the second figure specifies the transformation from the concrete syntax into instances of the metamodel classes, which are the nodes of the abstract syntax tree (the intermediate representation described in the *Compiler Overview*). The notation used to

specify the transformations is presented in the appendix § E. Both figures must be considered normative.

The constraints of each CML metamodel element are described on its own section. These sections refer to a `verbatim` figure, which contains the OCL [3] invariants (and its definitions) of the CML metamodel element in question, and it must be considered normative. Each invariant has a name in the format `inv_name` so that it can be referred by the compiler's error messages and users. Derived properties may also be defined before the constraints in order to simplify the constraint expressions. The appendix § D presents all the constraint rules in a single listing.

All metamodel elements referred by one of the descriptions defined above (definitions, examples, etc.) are emphasized in *italic*. If the descriptions of a CML metamodel element refer to another metamodel element, the corresponding chapter or section defining the other element is provided in parenthesis, like so (§ 3).

Some sections may not follow the structure defined above. These normally provide additional semantic information in plain English, which cannot be described using the notations presented above.

Part II

Conceptual Modeling

Four

Concepts

A *concept* in CML represents anything that has a coherent, cohesive and relevant meaning in a domain. In the ER [1] metamodel, it corresponds to an *entity set* (or an *entity type*); in UML [4], to a *class*. The CML *concept* differs, however, from the UML *class*, because it has only *properties* (§ 5), while the UML *class* may also have *operations*.

4.1 Example

Listing 4.1 presents some examples of *concepts* declared in CML. As shown, a *concept* may have zero or more *properties* (§ 5), and a *property* may optionally declare a *type* (§ 31, **??**). Also, as shown in the concept **EBook** of the example, a *concept* may specialize (§ 10) another *concept*.

4.2 Syntax

Listing 4.2 specifies the syntax used to declare a *concept*. The **concept** keyword is followed by a NAME. Optionally, a list of other NAMEs may be enumerated, referring to other *concepts* that are generalizations (§ 10) of the declared *concept*. A list of *properties* (§ 5) may be declared under the **concept** block. And the **abstract** keyword may precede the **concept** keyword, making a *concept* abstract (§ 11).

Figure 4.1 presents the *Concept* metaclass in an EMOF [5] class diagram, and listing 4.3 specifies the *concept* transformation from its concrete syntax to its abstract syntax. For each *concept* parsed by the compiler, an instance of the *Concept* class will be created, and its properties will be assigned according to parsed information:

```
// Empty concept:
@concept Book;

// Property without a type:
@concept TitledBook
{
    title;
}

// Property with the String type:
@concept StringTitledBook
{
    title: String;
}

// Specializing another concept:
@concept Ebook: Book;
```

Listing 4.1: Concept Examples

```
conceptDeclaration returns [Concept concept]:
    ABSTRACT? 'concept' NAME
    (':' generalizations)?
    (';' | propertyList);

generalizations:
    NAME (',' NAME)*;
```

Listing 4.2: Concept Concrete Syntax

- *name*: assigned with the value of the terminal node NAME.
- *abstract*: set to *true* if the **abstract** keyword is found before the **concept** keyword; otherwise, set to *false*.
- *elements*: an *ordered set* referencing all *properties* parsed in the **concept** block.
- *generalizations*: an *ordered set* referencing all *concepts* whose NAMEs were enumerated in the *GeneralizationList*.

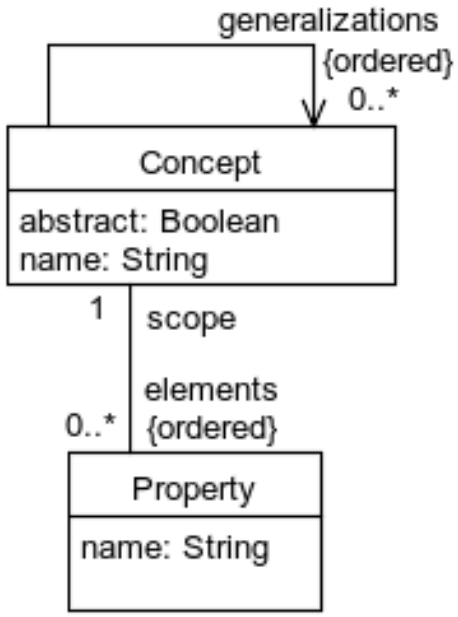

Figure 4.1: Concept Abstract Syntax

```
node Concept:
     'abstract'?
     'concept' NAME
     (':' GeneralizationList)?
     (';' | PropertyList)
{
     name = NAME;
     abstract = 'abstract'?;
     elements = PropertyList.Property*;
     generalizations = for name in GeneralizationList.NAME*
          | yield Model.concept[name];
}

node GeneralizationList: NAME (',' NAME)*;
```

Listing 4.3: Concept AST Instantiation

```
context Concept

inv unique_concept_name:
    parent.concepts
        ->select(c| c != self and c.name = self.name)
        ->isEmpty()
```

Listing 4.4: Concept Constraints

4.3 Constraints

Listing 4.4 presents the invariants of the *Concept* metaclass:

- *unique_concept_name*: Each *concept* must have a unique NAME within its *module* (§ 43).

Five

Properties

A *property* in CML may hold *values* of *primitive types* (§ 31), in which case they correspond to *attributes* (§ 6) on the ER [1] and UML [4] metamodels. Or they may hold references, or *sequences* of references (**??**), linking to instances of other *concepts* (§ 4), in which case they correspond to a *relationship* on the ER metamodel, or to *associations* (§ 8) on the UML metamodel.

5.1 Example

Listing 5.1 shows some *properties* declared in CML. As shown, a *property* may be an *attribute* (§ 6) of a *primitive type* (§ 31), or represent the *role* (or *end*) of an *association* (§ 8).

```
-- Attributes of primitive types:
@concept Book
{
    title: String;
    quantity: Integer;
}

-- Role in unidirectional association:
@concept Order
{
    customer: Customer;
}
```

Listing 5.1: Property Examples

```
propertyList:
    '{' (propertyDeclaration ';')* '}';

propertyDeclaration returns [Property property]:
   DERIVED? NAME (':' typeDeclaration)? ('=' expression)?;

DERIVED: '/';
```

Listing 5.2: Property Concrete Syntax

```
node PropertyList: '{' (Property ';')* '}';

node Property: '/'? NAME (':' Type)? ('=' STRING)?
{
    name = NAME;
    derived = '/'?;
    value = unwrap(STRING?);
    type = Type?;
}
```

Listing 5.3: Property AST Instantiation

5.2 Syntax

Listing 5.2 specifies the syntax used to declare a *property*. The NAME is followed by a *typeDeclaration* (§ 31 and **??**). Optionally, an *expression* (§ 12) may be specified in order to set the initial value.

Figure **??** presents the *Property* metaclass in an EMOF [5] class diagram of the CML metamodel, and listing 5.3 specifies the transformation from the *property* concrete syntax to its abstract syntax. For each *property* parsed by the compiler, an instance of the *Property* class will be created, and its properties will be assigned according to parsed information:

- *name*: assigned with the value of the terminal node NAME.
- *type*: if *typeDeclaration* is provided, *type* is set with the instance of the *Type* class matching the *typeDeclaration*.
- *expression*: if provided, it contains the instance of the *Expression* class matching the parsed *expression*.

```
context Property
inv unique_property_name:
    self.scope.properties
        ->select(p| p != self and p.name = self.name)
        ->isEmpty()

context Property
inv property_type_specified_or_inferred:
    type->notEmpty() or expression->notEmpty()

context Property
inv property_type_assignable_from_expression_type:
    type->notEmpty() and expression->notEmpty() implies
        type.isAssignableFrom(expression.type)
```

Listing 5.4: Property Constraints

5.3 Constraints

Listing 5.4 presents the invariants of the *Property* metaclass:

- *unique_property_name*: Each *property* must have a unique NAME within its *concept* (§ 4).
- *property_type_specified_or_inferred*: Either the *property* explicitly defines a *type* or it defines an *expression*, from which the type is inferred. That is required for both regular, slot-based *properties* (which may provide an *initialization expression*) and *derived properties* (which may have an *expression* defining the derivation).
- *property_type_assignable_from_expression_type*: When both a *type* and *expression* are defined for a *property*, the *type* inferred from the *expression* should be assignable to the declared *type*. That is required for both regular, slot-based *properties* (which may provide an *initialization expression*) and *derived properties* (which may have an *expression* defining the derivation).

Six

Attributes

In CML, *attributes* are *properties* (§ 5) of *primitive types* (§ 31). They correspond to the *Attribute* metaclass in the ER [1] metamodel; in the UML [4] metamodel, to the association *attribute* between the metaclass *Class* and the metaclass *Property*.

Attributes serve as a *slot* that holds a value of the specified *primitive type*. An initial value may be specified as an *expression* (§ 12). Some *attributes*, however, may be continuosly derive their *value* from an *expression* (not only initially), in which case they are called *derived attributes* (§ 7).

While initial values are only set when a *concept* (§ 4) is instantiated, the value of *derived attributes* is always evaluated from the given *expression*, and they cannot be set any other way.

6.1 Example

Listing 6.1 presents some examples of *attributes* declared in CML. As shown, the attribute **a** is a regular attribute definition that specifies the *primitive type* (§ 31) of the values that can be held by the *attribute*'s slot. The attribute **b** is an example showing how an *attribute* can be defined with an initial value. As shown by the attribute **c**, an attribute may be derived from an *expression* that refers to other *attributes*. In order to differentiate *attributes* with initial values from *derived attributes*, a forward slash ("/") prefixes the name of the latter. Attributes **d** and **e** are examples where the type of the attribute, instead of being specified, is inferred from the given *expression*. Type inference is possible for both regular, slot-based *attributes* and *derived attributes* that provide an *expression*.

```
@concept Attributes
{
    -- Attribute with a slot for values of a primitive type:
    a: Integer;

    -- An attribute with an initial value:
    b: String = "initial value";

    -- A derived attribute:
    /c: Decimal = 2.0 * a;

    -- An attribute with type inferred from its initial value:
    d = 3; -- Inferred as Integer based on constant "3"

    -- A derived attribute with type inferred from expression:
    /e = 2.0d * a; -- Inferred as Double based on "2.0d * a"
}
```

Listing 6.1: Attribute Example

6.2 Syntax

Listing 5.2 specifies the syntax used to declare any kind of *property* (§ 5), including *attributes*. The NAME of an *attribute* is followed by a *typeDeclaration* of a *primitive type* (§ 31). Optionally, an *expression* (§ 12) may be specified in order to set the initial value. A *derived attribute* must be prefixed with the forward-slash character, as specified by DERIVED, in which case the given *expression* defines the value of the *attribute* at all times.

Since an *attribute* in CML is just a *property* (§ 5) with *primitive types* (§ 31), the *property* metaclass in the CML metamodel is used to represent *attributes*. Figure **??** presents the *property* metaclass in an EMOF [5] class diagram, and listing 5.3 specifies the *property* transformation from its concrete syntax to its abstract syntax.

Seven

Derived Attributes

A *concept* in CML may have *attributes* (§ 6) that do not hold specific *values*, but instead provide a *value* derived from an *expression* (§ 12). These are called *derived attributes*. Unlike an *expression* used to initialize a *non-derived attribute*, the expression of a derived attribute is evaluated every time the attribute is queried.

In the UML [4] metamodel, the *Property* metaclass has a meta-attribute named *isDerived*, which determines whether an *attribute* is derived or not. A *derived attribute* in UML may be defined using a OCL [3] constraint; while CML has *expressions* as part of the language.

The ER [1] metamodel, in its original form, does not allow for the differentiation of *derived attributes* as part of an *entity set*, but it is possible to define *retrieval operations* whose results would equal to *values* of *derived properties* in CML. It can be said, however, that ER, by defining an *attribute* as a function from the *entity set* to the *value set*, does not prescribe that all *attributes* are memory-based, nor does it prevent the definition of an *attribute* function as an *expression*.

The CML metamodel and its syntax, on the other hand, define whether an *attribute* is memory-based (a *non-derived attribute*) or it is derived from an *expression* (a *derived attribute*).

7.1 Example

Figure **??** presents two examples of *derived attributes* declared in CML. As shown, the attribute **c** is derived from an *expression* that refers to other *attributes*. In order to differentiate *attributes* with initial values, such as **b**, from *derived attributes*, such as **c**, a forward slash ("/") prefixes the name of the latter. The attribute **e** is an example of a *derived attribute* where the type is inferred from the given *expression*, instead of being specified.

7.2 Syntax

Figure **??** specifies the syntax used to declare any kind of *property* (§ 5), including *derived attributes*. A *derived attribute* must be prefixed with the forward-slash character, as specified by DERIVED, in which case the given *expression* provides the value of the *attribute* every time it is fetched.

The *property* metaclass in the CML metamodel is used to represent *attributes*. Figure **??** presents the *property* metaclass in an EMOF [5] class diagram, and figure **??** specifies the *property* transformation from its concrete syntax to its abstract syntax. The *derived* property of the *Property* metaclass defines whether the *attribute* is derived or not.

Eight

Associations

In CML, an *association* represents a relation between two *concepts* (§ 4), where a reference to an *instance* of each *concept* is found in every tuple that is part of the relation. When *concepts* have an *association* between themselves, its *instances* are linked in such way that it is possible to access an *instance* of one *concept* from an *instance* of the other *concept*.

The UML [4] metamodel has a metaclass named *Association* that has *Property* instances, whose *types* are the *Class* instances that are part of the *association*. In UML, the name of each *Property* instance in the *Association* metaclass is known as the *role* of the corresponding *Class* in the *association*.

On the CML metamodel, on other hand, the *Association* metaclass is only needed when it is necessary to define *bidirectional associations*, whose *links* are accessible from either *association end*. For *unidirectional associations*, where only one *association end* is accessible, only a *property* is defined in the source *concept*, making its *type* the target *concept*.

On the ER [1] metamodel, each *association* is known as a *relationship set*, and each tuple in this set is called a *relationship*. Unlike CML and UML, the tuples in a *relationship set* of an ER model can be queried directly, and no notion of *property* is required as part of the *entity type* in order to access those *relationships*.

As it is case for *attributes* (§ 6), *associations* in CML can also be derived from other *associations* (just as well as in UML); they are called *derived associations* (§ 9).

8.1 Example

Listing 8.1 presents some examples of *associations* declared in CML. The concept **Vehicle** contains the property **driver**, which may optionally refer to an instance of **Employee**, meaning that a **driver** may or may not be assigned to a single **Vehicle**. The concept **Vehicle** also has the property **owner**, which always refers to an instance of **Organization**, meaning that an **owner** must always be assigned to each instance of **Vehicle**. Similarly, the concept **Employee** has the property **employer**, which must always be assigned to an instance of **Organization**.

Just below the declaration of **Organization**, we observe an association named **Employment**, which enumerates two *properties*: the first is **employer** from the concept **Employee**; the second is **employees** from the concept **Organization**. What this *association* implies is a correspondence between these two properties. Every time a reference to an instance of **Organization** is assigned to the slot **employer** of an instance of **Employee**, a reference to this same instance of **Employee** must be assigned to the slot **employees** of the **Organization** instance. However, since the *type* of **employees** in the concept **Organization** is a sequence (**??**) of **Employee** instances, the reference to the instance of **Employee** will actually be appended to the sequence being held by the slot **employees** of the concept **Organization**, and maintained along with the other **Employee** instances already found in the sequence. Thus, the association **Employment** actually characterizes a *bidirectional association*.

The association **VehicleOwnership** is another example of a *bidirectional association*; in this case, between **Vehicle**'s **owner** property and **Organization**'s **fleet** property. It can be noticed, though, in this second *bidirectional association*, that the *types* of the *properties* are declared along with their names; such a *type* declaration, in the *association* declaration, is optional in CML, but must match the original *property* declaration under the *concept* declaration, if present.

The **driver** property in the concept **Vehicle** is a different case, since this *property* does not participate in any *association* declaration on listing 8.1. That's because there is no corresponding *property* in the concept **Employee** representing the other end of the *association*. As such, the property **driver** is representing the source end of a *unidirectional association*.

The property **drivers** in the concept **Organization** is *derived association* (§ 9).

```
@concept Vehicle
{
    plate: String;
    driver: Employee?;
    owner: Organization;
}

@concept Employee
{
    name: String;
    employer: Organization;
}

@concept Organization
{
    name: String;
    employees: Employee*;
    fleet: Vehicle*;
    /drivers = fleet.driver;
}

@association Employment
{
    Employee.employer;
    Organization.employees;
}

@association VehicleOwnership
{
    Vehicle.owner: Organization;
    Organization.fleet: Vehicle*;
}
```

Listing 8.1: Association Example

```
associationDeclaration
    returns [Association association]:
    'association' NAME
    '{' (associationEndDeclaration ';')* '}';

associationEndDeclaration
    returns [AssociationEnd associationEnd]:
    conceptName=NAME '.' propertyName=NAME
    (':' typeDeclaration)?;
```

Listing 8.2: Association Concrete Syntax

8.2 Syntax

The concrete syntax used to declare an *association* in CML is specified by listing 8.2. First, the **association** keyword is followed by a NAME. Then, a list of *association ends* are declared under the **association** block. For each declaration of an *association end*, The **conceptName** and **propertyName** are optionally followed by a **typeDeclaration**.

The *Association* metaclass is presented in the EMOF [5] class diagram of figure **??**, and its instantiation from the concrete syntax is specified by listing 8.3. For each parsed *association*, an instance of the *Association* metaclass will be created, and its meta-properties will be assigned according to parsed information:

- *name*: assigned with the value of the token NAME.
- *members*: an *ordered set* referencing all *associationEnd* instances parsed in the **association** block.

8.3 Constraints

The invariants of the metaclasses *Association* and *AssociationEnd* are specified by listing 8.4:

- *association_end_property_found_in_model*: Each *association end* enumerated under an *association* must correspond to a *property* of the same name under the specified *concept*.

```
node Association:
    'association' NAME '{' (AssociationEnd ';')* '}'
{
    name = NAME;
    members = AssociationEnd*;
}

node AssociationEnd:
    conceptName=NAME '.' propertyName=NAME (':' type=Type)?
{
    concept = Model.concepts
                ->select(concept.name = conceptName)
                ->first()

    property = concept.allProperties
                    ->select(property.name = propertyName)
                    ->first()
}
```

Listing 8.3: Association AST Instantiation

- *association_end_type_matches_property_type*: If a *type* is specified for a given *association end*, its *name* and *cardinality* must match the *type* of the corresponding *property*.
- *association_must_have_two_association_ends*: An *association* must have exaclty two *association ends*, since only *binary associations* are supported in CML.
- *association_end_types_must_match*: The *concept* of one *association end* must correspond to the *type* of the *property* of the other *association end*, and vice-versa.

8.4 Translation

```
context AssociationEnd
inv association_end_property_found_in_model:
    concept->notEmpty() and property->notEmpty()

context AssociationEnd
inv association_end_type_matches_property_type:
    self.propertyType->notEmpty() and
    self.property->notEmpty() implies
        self.propertyType.name = property.type.name and
        self.propertyType.cardinality = property.type.cardinality

context Association
inv association_must_have_two_association_ends:
    associationEnds->count() = 2

context Association
def: first = associationEnds->first()
def: last = associationEnds->last()
inv association_end_types_must_match:
    associationEnds->count() = 2 and
    first->notEmpty() and last->notEmpty() and
    first.concept->notEmpty() and
    first.property->notEmpty() and
    last.concept->notEmpty() and
    last.property->notEmpty() implies
        first.concept.name = last.property.type.name and
        last.concept.name = first.property.type.name

inv property_must_be_part_of_single_association:

inv no_associations_of_primitive_types:
```

Listing 8.4: Association Constraints

Nine

Derived Associations

Derived associations are declared by *derived properties* (§ 5), which associate a *source-concept* to a *target-concept* via an *expression* (§ 12). In order to obtain a reference to a *target-concept* (§ 4), it is necessary to evaluate the *expression* of the *derived property*. Among the possible *sub-expressions* that are part of the *expression* of the *derived association*, there are *path expressions* (§ 22) of other *associations*, which server as the base for the *derived association*.

Ten

Generalization / Specialization

A *concept* (§ 4) in CML may be generalized by another *concept*. In other words, a *concept* may be considered a specialization of another *concept*. Generalized *concepts* have *properties* (§ 5) that apply to a larger set of instances, while specialized *concepts* have *properties* that only apply to a subset of those instances.

In the UML [4] metamodel, such generalization/specialization relationship between *classes* is known as *generalization*, which is the name of the metaclass in the UML metamodel. The original version of the ER [1] metamodel lacked this kind of relationship between *entity types*.

10.1 Example

Figure 10.1 presents some examples of generalization/specialization relationships declared in CML. As shown, a *concept* (§ 4) may specialize zero or more other *concepts*. The latter are called the generalizations, while the former is called the specialization. A generalization, such as **Shape**, may define *attributes* (§ 6), such as **color** and **area**, or also the *roles* in *unidirecional associations* (§ 8). Both *attributes* and *roles* are *properties* (§ 5) shared among all its specializations. Some of these *properties* may be redefined by the some of the specializations, as it is the case with the *area* property, which is redefined by **Rectangle**, **Rhombus** and **Square**. Some specializations may also define new *properties*, such as **width** and **height** in **Rectangle**, which characterize only instances of this specialization. A *concept* may be a specialization of two or more other *concepts*, as seen with **Square**, which specializes both **Rectangle** and **Rhombus**, and thus can redefine *properties* of both generalizations. If a *property* has been defined by more than one generalization, then it must be redefined by the

specialization in order to resolve the definition conflict, which is the case with **area** in **Square**. If a redefinition suitable for both generalizations is unattainable, it may be an indication that either the specialization or the generalizations are unsound from the domain's prospective.

10.2 Syntax

Figure **??** specifies the syntax used to declare a *concept* (§ 4), and in turn its generalizations. A list of NAMEs may be enumerated after the declared *concept*'s NAME, referring to other *concepts* that this concept is a specialization of.

Figure 4.1 presents the *Concept* metaclass in an EMOF [5] class diagram of the CML metamodel, and figure **??** specifies the *concept* transformation from its concrete syntax to its abstract syntax. There is a unidirecional association in the *Concept* class that keeps track of the generalization/specialization relationships, which is named *generalizations*. It is an *ordered set* referencing all *concepts* whose NAMEs were enumerated in the *GeneralizationList* of the declared *concept*.

10.3 Constraints

Figure 10.2 presents the invariants of the *Concept* and *Property* classes related to *generalizations*:

- *not_own_generalization*: A *concept* (§ 4) may not be listed on its own *GeneralizationList*, nor on the *GeneralizationList* of its direct or indirect generalizations.
- *compatible_generalizations*: The *generalizations* of a *concept* must all be compatible between themselves, that is, no two *generalizations* may have a *property* with the same name but a different type.
- *generalization_compatible_redefinition*: A *property* may only be redefined with the same type defined in the *generalizations*.
- *conflict_redefinition*: A *concept* is required to redefine a *property* that has been defined by two or more of its *generalizations* in order to resolve the definition conflict. That is required only if the *property* has been initialized or derived in at least one of the *generalizations*. Otherwise, the redefinition is not required.

```
-- Generalization of Circle and Rectangle:
@concept Shape
{
    -- Specializations below share the color attribute as-is:
    color: String;

    -- Specializations below redefine the area attribute:
    area: Double;
}

-- Specialization of Shape:
@concept Rectangle: Shape
{
    -- New attributes that characterize a rectangle:
    width: Double;
    height: Double;

    -- Redefinition of the area attribute:
    /area = width * height;
}

-- Another specialization of Shape:
@concept Rhombus: Shape
{
    -- Diagonal attributes that characterize a rhombus:
    p: Double;
    q: Double;

    -- Another redefinition of the area attribute:
    /area = (p * q) / 2.0d;
}

-- Specialization of both Rectangle and Rhombus:
@concept Square: Rectangle, Rhombus
{
    -- Only attribute needed to characterize a square:
    side_length: Double;

    -- Redefinitions of Rectangle's attributes:
    /width = side_length;
    /height = side_length;

    -- Redefinitions of Rhombus' attributes:
    /p = side_length * 1.41421356237d; -- square root of 2
    /q = p;

    -- Required to redefine area in order to resolve conflict
    -- between Rectangle's area and Rhombus' area:
    /area = side_length ^ 2.0d;
}
```

Figure 10.1: Generalization Examples

```
context Concept::all_generalizations: Set(Concept)
derive:
    generalizations->closure(generalizations)

context Concept::all_properties: Set(Property)
pre:
    all_generalizations->excludes(self)
derive:
    elements->union(
        generalizations.all_properties->select(p1|
            not elements->exists(p2| p1.name == p2.name)
        )
    )

context Concept::generalization_pairs
                : Set(Tuple(left: Concept, right: Concept))
derive:
    generalizations->collect(g1|
        generalizations
            ->select(g2| g1 != g2)
            ->collect(g2| Tuple { left: g1, right: g2 })
    )->flatten()

context Concept::generalization_property_pairs
                : Set(Tuple(left: Property, right: Property))
derive:
    generalization_pairs->collect(pair|
        pair.left.all_properties->collect(p1|
            pair.right.all_properties
                ->select(p2| p1 != p2 and p1.name = p2.name)
                ->collect(p2| Tuple { left: p1, right: p2 })
        )->flatten()
    )->flatten()

context Concept
inv not_own_generalization:
    all_generalizations->excludes(self)

context Concept
inv compatible_generalizations:
    generalization_property_pairs
        ->forAll(
            left.type.name = right.type.name and
            left.type.cardinality = right.type.cardinality
        )

context Concept
inv conflict_redefinition:
    generalization_property_pairs
        ->select(left.type = right.type)
        ->select(left.derived or left.expression->notEmpty() or
                 right.derived or right.expression->notEmpty())
        ->forAll(self.elements->exists(name = left.name))

context Property
inv generalization_compatible_redefinition:
    self.scope.generalizations.all_properties
        ->select(property| self.name = property.name)
        ->forAll(property|
            self.type.name = property.type.name and
```

Eleven

Abstractions

An *abstraction* in CML is a *concept* (§ 4) that cannot create instances on its own, but instead serves as a *generalization* (§ 10) for other *concepts*, which in turn can create their own instances. Thus, all instances of an *abstraction* are first instances of its *specializations*.

In CML, an *abstraction* may also define a *derived property* (§ 5) without providing an *expression* (§ 12) in its definition; such *properties* are called *abstract properties*.

CML's support for *abstractions* matches UML's [4], which allows the declaration of *abstract classes* by setting the *isAbstract* attribute of a *Class* instance to *true*. UML also allows the declaration of *abstract attributes* and *abstract operations*.

The original version of the ER [1] metamodel, however, as a consequence of lacking the *generalization/specialization* relationship, has not considered the notion of *abstract entities*.

11.1 Example

Listing 11.1 presents an example of an *abstract concept* declared in CML. As shown, the concept **Shape** is tagged as *abstract*, and as such no direct instances of *Shape* are ever instantiated. As an *abstract concept*, **Shape** can define *abstract properties*, like **area**, which is just a *derived property* (§ 5) without an *expression* (§ 12). An *abstract concept* may also define concrete *properties*, such as **color** in **Shape**. The concept **Circle** is a *especialization* of **Shape** that must redefine the property **area** (and provide an *expression*) if it is to be considered a *concrete concept*. As a *concrete concept*, **Circle** may have direct instances, which are in turn instances of *Shape* as well. **Circle** may also redefine *concrete properties* of **Shape**, like **color**, but the redefinition is not a requirement in this case. In **UnitCircle**, we can observe that

the redefinition of an *abstract property*, such as **area**, may be made *concrete*; meaning it does not need to be redefined as a *derived property*. The converse situation is also allowed in CML, where a *concrete property* is redefined by as a *derived property*, as illustrated with the property **radius** in **UnitCircle**.

11.2 Syntax

Listing 4.2 specifies the syntax used to declare a *concept* (§ 4) in CML. It shows that a *concept* may be tagged with the **abstract** keyword in order to convey it as an *abstract concept*. Listing 5.2 specifies the syntax used to declare a *property* (**??**) in CML. It shows that a *property* may be prefixed with a forward slash ("/") in order to mark it as a *derived property*. If the optional **expression** is not provided, the property is then considered an *abstract property*.

Figure 4.1 presents the *concept* metamodel in an EMOF [5] class diagram, and listing 4.3 specifies the *concept* transformation from its concrete syntax to its abstract syntax. There is a **Boolean** attribute named **abstract** in the *Concept* class that determines whether a *concept* is *abstract* or not.

11.3 Constraints

Listing 11.2 presents the invariants of the *Concept* and *Property* classes in CML's EMOF [5] metamodel related to *abstract concepts*:

- *abstract_property_redefinition*: A *concrete concept* must redefine concretely all *abstract properties* of its *generalizations*.
- *abstract_property_in_abstract_concept*: Only *abstract concepts* may have *abstract properties*.

```
-- As an abstract concept,
-- no direct instances of Shape are ever created.
@abstraction Shape
{
    -- A derived property without an expression
    -- is considered abstract.
    -- Only abstract concepts may have abstract properties.
    /area: Double;

    -- Abstract concepts may also have concrete properties:
    color: String;
}

-- All instances of Circle are in turn instances of Shape.
@concept Circle: Shape
{
    radius: Double;

    -- In order to be considered a concrete concept,
    -- Circle must redefine the abstract properties
    -- inherited from Shape.
    /area = 3.14159d * radius ^ 2;

    -- Circle may also redefine concrete properties of Shape.
    -- However, the redefinition is not required in this case.
    color = "Blue";
}

@concept UnitCircle: Circle
{
    -- Observe below that the redefinition of
    -- an abstract property may be concrete;
    -- that is, it does not have to be derived
    -- as it was done in Circle.
    area = 3.14159d;

    -- In the case above, however,
    -- it is desirable to redefine "area" as a derived property,
    -- in order to guarantee area's value cannot be modified
    -- after the instantiation of UnitCircle.
    -- This is done with the redefinition of "radius" below.
    -- Notice that, in Circle, radius was concrete,
    -- but its redefinition below makes it derived.
    -- That's allowed in CML just as the other way around,
    -- as it was done with "area" above.
    /radius = 1.0d;
}
```

Listing 11.1: Abstract Concept Example

```
context Property::abstract: Boolean
derive:
    self.derived and self.expression->isEmpty()

context Property::concrete: Boolean
derive:
    not self.abstract

context Concept
inv abstract_property_redefinition:
    self.concrete implies
        self.generalizations.all_properties
            ->select(abstract)
            ->forAll(p1|
                self.properties
                    ->select(p2| p1.name = p2.name)
                    ->reject(abstract)
                    ->notEmpty()
            )

context Property
inv abstract_property_in_abstract_concept:
    self.abstract implies self.scope.abstract
```

Listing 11.2: Abstract Concept Constraints

Part III

Expressions

Twelve

Expressions

An *expression* in CML is used to compute *values* (§ 31) or *references* (§ 36) to *concept* instances (§ 4). They are used to initialize or derive *properties* (§ 5). On the UML [4] metamodel, it corresponds to the *Expression* metaclass; in OCL [3], to *OclExpressionCS*.

CML *expressions* are designed to provide the same level of expressivity provided by OCL *expressions*, but the CML syntax varies from OCL's; sintactically, they differ especially on OCL's *collection operations*, which correspond to *comprehensions* (§ 25) in CML.

CML allows the use of *operators* in *expressions*. The categories of *operators* in CML are: *arithmetic operators* (§ 14), *relational operators* (§ 16), *logical operators* (§ 15), *referential operators* (§ 17), *type-checking operators* (§ 19) and *type-casting operators* (§ 20). Most of the *operators* in CML are infixed, with just three of them (`not`, `+` and `-`) being used as a prefix. The use of parenthesis is allowed to estabilish precedence.

Besides the *operators* listed above, CML also offers the following *expressions*: *paths* (§ 22), *invocations* (§ 23), *lambdas* (§ 24) and *comprehensions* (§ 25). They are all presented in the following chapters.

12.1 Examples

Listing 12.1 has some examples of CML *expressions*. As shown, there are different types of expressions: *literal values* (§ 13), *prefix expressions* (**??**), *infix expressions* (**??**), *conditional expressions* (§ 18), *path expressions* (§ 22) and *query expressions* (**??**).

```
@concept Expressions
{
    -- Literal Values:
    c: String = "SomeString";
    d: Integer = 123;

    -- Prefix Expression:
    minus_sign = -2;

    -- Infix Expressions:
    addition = 1 + 2;
    equality = 3 == 3;
    boolean_expr = q and p;

    -- Conditional Expression:
    if_then_else = if a > 0 then a else b;

    -- Path Expression:
    path = somePath.bar;

    -- Query Expression:
    select_query = items | select: name == "this";
}
```

Listing 12.1: Expression Example

12.2 Syntax

Listing 12.2 specifies the syntax of all kinds of *expressions* in CML. It also lists them in their order of precedence.

Observe that the grammar in listing 12.2 has left recursions (and thus it is ambiguous). However, ANTLR [6] resolves the ambiguity based on the order in which the *expression* alternatives are listed, and so the order in listing 12.2 defines the precedence among the operators for CML.

Also, according to ANTLR, and as required by CML, all *expressions* in the grammar are left-to-right associative, except for the *exponentiation expression*, which is right-to-left associative, as defined by the **<assoc=right>** clause.

Figure 12.1 presents the *Expression* metaclass in an EMOF [5] class diagram. For each kind of *expression* parsed by the compiler, an instance of an *Expression* subclass will be created, and its properties will be assigned according to parsed information:

- *kind*: a *String* value matching the *Expression* subclass; for example,

```
expression returns [Expression expr]
    : literalExpression
    | pathExpression
    | operator=('+' | '-' | NOT) expression
    | <assoc=right> expression operator='^' expression
    | expression operator=('*' | '/' | '%') expression
    | expression operator=('+' | '-') expression
    | expression operator=('<' | '<=' | '>' | '>=') expression
    | expression operator=('==' | '!=') expression
    | expression operator=AND expression
    | expression operator=OR expression
    | expression operator=XOR expression
    | expression operator=IMPLIES expression
    | IF cond=expression
      THEN then=expression
      ELSE else_=expression
    | queryExpression
    | '(' inner=expression ')';

queryExpression returns [Expression expr]
    : pathExpression
    | joinExpression
    | queryExpression '|' transformDeclaration;

joinExpression returns [Join join]:
    FOR enumeratorDeclaration (',' enumeratorDeclaration)*;

enumeratorDeclaration:
    var=NAME IN pathExpression;

transformDeclaration returns [Transform transform]:
    (FROM var=NAME '=' init=expression)?
    operation=
        ( SELECT    | REJECT
        | YIELD     | RECURSE
        | INCLUDES  | EXCLUDES
        | EVERY     | EXISTS
        | REDUCE
        | TAKE      | DROP
        | FIRST     | LAST
        | COUNT     | SUM       | AVERAGE
        | MAX       | MIN
        | REVERSE)
    suffix=(UNIQUE | WHILE)?
    expr=expression?;
```

Listing 12.2: Expression Concrete Syntax

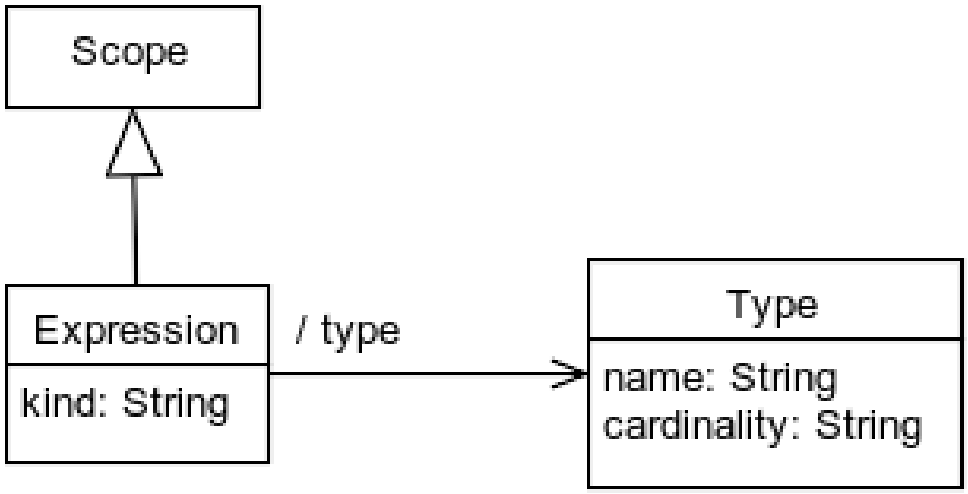

Figure 12.1: Expression Abstract Syntax

for the *Literal* subclass, **kind = "literal"**.

- *type*: a derived attribute that computes the *Type* of the *expression*; each *Expression* subclass will do its own *Type* computation by providing its own definition for this derived attribute.

Thirteen

Literal Expressions

For each one of the primitive types (§ 31), there are corresponding *literal expressions* that can represent their values. In CML, as it is the case with UML [4], OCL [3] and ER [1], each *primitive type* may considered a mathematical *set*. A *literal value* allowed by a *type* (§ 30) corresponds to an *element* belonging to this *set*.

CML is able to infer the *primitive type* of a *property* (§ 5) based on the syntax of a *literal expression*. Each *primitive type* in CML has a unique syntax. In contrast, OCL has the same *literal* syntax for all *numeric types*, and thus cannot infer a *primitive type* from a *literal expression*.

13.1 Examples

Table 13.1 displays examples of *literal expressions* in CML.

Type	Example
String	`"Hello!\n"`
Boolean	`true`
Integer	`123456`
Decimal	`1234.567`
Byte	`127b`
Short	`1234s`
Long	`12345671`
Float	`1234.567f`
Double	`1234.567d`

Table 13.1: Literal Expressions for Primitive Types

13.2 Syntax

Table 13.2 displays the syntax of *literal expressions* in CML.

Type	Syntax
String	`'"' ('\\'[btnr"\\] | . )*? '"'`
Boolean	`'true' | 'false'`
Integer	`('0'..'9')+`
Decimal	`('0'..'9')* '.' ('0'..'9')+`
Byte	`('0'..'9')+ 'b'`
Short	`('0'..'9')+ 's'`
Long	`('0'..'9')+ 'l'`
Float	`('0'..'9')* '.' ('0'..'9')+ 'f'`
Double	`('0'..'9')* '.' ('0'..'9')+ 'd'`

Table 13.2: Literal Expressions for Primitive Types

Fourteen

Arithmetic Expressions

Arithmetic expressions are composed by the *arithmetic operators*, which only accept as operands the *expressions* of *numeric* (§ 33) and *floating-point* (§ 34) types.

14.1 Examples

Table 14.1 displays examples of *arithmetic expressions* in CML.

Operator	Operation	Example
`^`	Exponentiation	`3 ^ 2`
`*`, `/`, `%`	Multiplication, Division, Modulo	`6 * 2 / 3 % 4`
`+`, `-`	Addition, Subtraction	`6 + 2 - 1`

Table 14.1: Examples using the Arithmetic Operators

14.2 Constraints

Table 14.2 shows the precende order and associativity of *arithmetic operators* in CML.

Operator	Operation	Associativity
^	Exponentiation	Right
*, /, %	Multiplication, Division, Modulo	Left
+, −*	Addition, Subtraction	Left

*The addition/subtraction operators may also be prefixed in the unary form.

Table 14.2: Arithmetic Operators in Precedence Order

All *arithmetic operators* are infixed, but the addition (+) and subtraction (−) operators may also be prefixed when used in the unary form. The associativity of the arithmetic operators is from left to right, except for the exponentiation operator (^), where it is from right to left.

A validation error should reported by the compiler if any of the *operands* of an *arithmetic expression* is inferred to be of a *type* other than *numeric* (§ 33) or *floating-point* (§ 34).

Fifteen

Logical Expressions

Logical expressions are composed by the *logical operators*, which only accept as operands the *expressions* (§ 12) of the *Boolean* type (§ 32).

15.1 Examples

Table 15.1 displays examples of *logical expressions* in CML.

Operator	Operation	Example
`not`	Negation	`not p`
`and`	Conjunction	`p and q`
`or`	Disjunction	`p or q`
`xor`	Exclusive Disjunction	`p xor q`
`implies`	Implication	`p implies q`

Table 15.1: Logical Operators in Precedence Order

15.2 Constraints

Table 15.2 shows the precende order of the *logical operators* in CML.

Operator	Operation
`not`	Negation
`and`	Conjunction
`or`	Disjunction
`xor`	Exclusive Disjunction
`implies`	Implication

Table 15.2: Logical Operators in Precedence Order

All logical operators are infixed, except for the negation operator (`not`), which is prefixed and unary. The associativity of the binary operators is always from left to right.

A validation error should reported by the compiler if any of the *operands* of a *logical expression* is inferred to be of a *type* other than *Boolean*.

Sixteen

Relational Expressions

Relational expressions are composed by the *relational operators*, which only accept as operands the *expressions* (§ 12) of a *relational type*, which include *String* (§ 35), the *numeric* (§ 33) and *floating-point* (§ 34) types, but exclude *Boolean* (§ 32) and *reference types* (§ 36).

16.1 Examples

Table 16.1 shows examples of the *relational expressions* in CML.

Operators	Operation	Example
<, <=, >, >=	Ordering	`3 < 2`
==, !=	Equality, Unequality	`6 != 3`

Table 16.1: Relational Operators

16.2 Constraints

Table 16.2 shows the precende order of the *relational operators* in CML.

Operators	Operation
<, <=, >, >=	Ordering
==, !=	Equality, Unequality

Table 16.2: Relational Operators

All *relational operators* are infixed. There is no associativity between them.

A validation error should reported by the compiler if any of the *operands* of a *relational expression* is inferred to be of a *type* other than *String* (§ 35), *numeric* (§ 33) or *floating-point* (§ 34).

Seventeen

Referential Expressions

In CML, *referential expressions* are composed by the referential operators, which only accept as operands the *expressions* (§ 12) resulting in references (§ 36). That excludes all *primitive types* (§ 31).

The *referential equality operator* (`is`) results in the `true` value if both operands reference the same instance. The *referential inequality operator* (`is`) results in `true` if the refecences do not point to the same instance.

17.1 Examples

Table 17.1 shows examples of *referential expressions* in CML.

Operator	Operation	Example
`===`	Referential Equality	`product === book`
`!==`	Referencial Inequality	`product !== book`

Table 17.1: Referential Operators

17.2 Constraints

The *referential operators* are infixed. There is no associativity between them.

A validation error should reported by the compiler if any of the *operands* of a *referential expression* is inferred to be of a *type* other than a *reference type* (§ 36).

Eighteen

Conditional Expressions

In CML, *conditional expressions* allow alternating between one or more *expressions* (§ 12) based on some *condition*, which is an *expression* of *Boolean* type (§ 32). The remaining operands of a *conditional expression* – the alternating *expressions* – may be of any type (§ 30), including the *primitive types* (§ 31) and *references* (§ 36).

The *conditional expressions* are divided in three categories:

- unary: only evaluates a single *expression* if the *condition* evaluates to `true`.
- binary: results in the evaluation of the first *expression* if the *condition* evaluates to `true`; otherwise, it results in the evaluation of the second *expression*.
- optional: the *condition* is implicit; it results in the first *expression* if it provides a value; else, it results in the second *expression* if it provides a value; otherwise, it results in *none*.

The resulting type of a *conditional expression* is based on the type of its *operands*. The cardinality is also based on the *operands*.

18.1 Examples

Table 18.1 displays examples of *conditional expressions* in CML.

Expression	Example
unary if	`10 if i > 2`
unary unless	`10 unless i <= 2`
binary if-then-else	`if i > 2 then 10 else 5`
binary if-else	`10 if i > 2 else 5`
conditional or	`item.book or? item.accessory`
conditional xor	`item.book xor? item.accessory`

Table 18.1: Conditional Expressions

18.2 Syntax

Table 18.2 displays the syntax of *conditional expressions* in CML.

Expression	Example
unary if	`<expr> if <cond>`
unary unless	`<expr> unless <cond>`
binary if-then-else	`if <cond> then <expr1> else <expr2>`
binary if-else	`<expr1> if <cond> else <expr2>`
conditional or	`<expr1> or? <expr2>`
conditional xor	`<expr1> xor? <expr2>`

Table 18.2: Syntax of Conditional Expressions

18.3 Constraints

In *conditional expressions*, the *condition* must be of *Boolean* type, and the *alternatives* must be of compatible types.

Nineteen

Type Checking

The *type-checking operator* (`is`) results in the `true` value if the instance referenced by the first operand is of the *type* (§ 30) specified by the second operand, or it is an *specialization* of such type. The *nagative type-checking operator* (`is`) results in `true` if the *type* does not match.

19.1 Examples

The table 19.1 shows examples of the *type-checking expressions*.

Operator	Operation	Example
`is`	Type-Checking	`product is Book`
`isnt`	Negative Type-Checking	`product isnt Book`

Table 19.1: Examples of Type-Checking Expressions

Twenty

Type Casting

There are three *type-casting operators* in CML. The *safe type-casting operator* (`as`) may only be applied when the casting is guaranteed to work and the resulting expression has the same cardinality as the original one. The *forced type-casting operator* (`as!`) may raise an exception at runtime if the actual type of the instance is not compatible with the expected type. The *conditional type-casting operator* (`as?`) automatically selects the compatible instances, never raising an expception at runtime.

20.1 Examples

The table 20.1 presents the *type-casting expressions*.

Oper.	Operation	Example
`as`	Safe Type-Casting	`products as Book*`
`as!`	Forced Type-Casting	`products as! Book`
`as?`	Condicional Type-Casting	`products as? Book?`

Table 20.1: Examples of Type-Casting Expressions

Twenty-one

String Concatenation

Expressions (§ 12) of the *primitive types* (§ 31), such as the *String* type (§ 35), may be combined with the ampersand (*&*) operator for string concatenation.

21.1 Constraints

Only an *expression* inferred to be of a *primitive type* (§ 31) may be used as an operand of a *string concatenation* expression. It is allowed that none of the operands are of *String* type. All operands are converted to *String* before concatenation.

The associativity of the concatenation operator is from left to right, just like the *arithmetic operators* (§ 14).

21.2 Type Inference

The resulting *expression* is always of the *String* type, regardless of the operand types.

21.3 Translation

The *operands* are concatenated using the string concatenation operation provided by the target language. If the *operand type* is not *String*, then it is evaluated and the resulting value is converted to a *String* value, which is then concatenated with the other *String* values. The conversion of the *operands* is dependent on the target programming language, but it is nor-

mally done using the standard methods, such as *Objects.toString()* in Java, or *str()* in Python.

Twenty-two

Path Expressions

In CML, *path expressions* allow accessing the *values* (§ 31) and *references* (**??**) of *properties* (§ 5) in instances of a *concept* (§ 4). Be those *properties* the *attributes* (§ 6) or the *association roles* (§ 8) of a *concept* instance, *path expressions* will traverse through each *property* in the path in order to find the intended *values* or *references*. They can also be applied to *lambda parameters* (§ 24).

Twenty-three

Invocation Expressions

In CML, *invocations* apply a *function* (§ 26) to a number of *expressions* (§ 12) – the arguments – that correspond to the *function parameters*. CML provides some *built-in functions* in the `cml_base` module. New functions may be defined via *template functions* (§ 27) or via *declared functions* (§ 28).

Twenty-four

Lambda Expressions

CML allows the use of *lambda expressions* as an argument of the *invocation* (§ 23) of a *function* (§ 26). CML provides some *built-in functions* in the `cml_base` module that accept a *lambda expression* as an argument.

Lambda expressions may be seen as an inline function definition. Just like *functions*, *lambda expressions* may have a number of *parameters* that are used in its inner expression.

CML also allows a parameterless *lambda expression* to have an undeclared, implicit parameter, which serves as its scope. This implicit parameter in the *lambda expression* is inferred from the *function* declaring the *lambda* as one of its parameters. *Functions* may declare a *lambda parameter* using a *function types* (§ 38).

Additionally, *lambda expressions* in CML are considered pure functions (just like *declared functions* § 28) given that the *invocations* performed by the inner expressions is also pure *functions* or *lambdas*.

Twenty-five

Comprehension Expressions

CML supports a kind of *expression* (§ 12) called *comprehension*, which is a flexible and expressive way to create new *sequences* (**??**) or to calculate *values* or *references* from existing ones.

The *comprehension expressions* in CML are similar to the set builder notation in mathematics, and also to *list comprehensions* in Python, or *for comprehensions* in Scala. Unlike those languages, in CML, *comprehensions* can be combined using the pipe (|) operator and *functions* (§ 29), making them more extensible.

Part IV

Functions

Twenty-six

Built-In Functions

CML provides *built-in functions* in the `cml_base` module. Most of them allow the processing of references (**??**) and sequences (**??**).

Most *built-in functions* are defined via *template functions* (§ 27) for each target programming language or technology. The *templates* that implement the *built-in functions* must ensure they are pure functions (just like *declared functions* § 28) so that their *invocations* from *declared functions* (§ 28) or *lambdas* (§ 24) do not invalidate their *purity*.

Twenty-seven

Template Functions

In CML, *template functions* allow the definition of a *function* in the target programming language via StringTemplate [7]. The declaration of a *template function* only defines the *function signature*, while the *function expression* is defined in StringTemplate for each target programming language.

Twenty-eight

Declared Functions

In CML, *functions* may be defined by a *signature* and a corresponding *expression* (§ 12). The *function signature* declares the *parameter* and the resulting *type* (§ 30). Such a *function* is translated to a target programming language much like a *derived attribute* (§ 7), except that it does not belong to any particular *concept* (§ 4) and it may be invoked by any *expression*, in any scope.

Functions declared in CML are pure functions, which means they have the following characteristics:

- they always evaluate to the same result given the same arguments;
- their evaluation does not cause any side-effects or state mutation.

If a *declared function* in its definition invokes other *functions*, then those *functions* (being them other *declared functions* or *template functions* § 27) must provide the same guarantees listed above in order for the *declared function* to be considered pure.

Twenty-nine

Comprehension Functions

Some *functions*, being *declared* (§ 28) or just *templates* (§ 27), may be used in *comprehension expressions* (§ 29) if they have a specific *signature*. They may have one or more *parameters* as long as the first *parameter* is named `seq` and it is of a *sequence type* (**??**).

Part V

Type System

Thirty

Types

CML is a statically-typed language. All its *properties* (§ 5) and *expressions* (§ 12) must have a *type* declared or inferred at compilation time. Additionally:

- the type of a *property* must be compatible with the type of its *expression*;
- the type of the *operands* in an *expression* must be compatible with its *operator*;
- the type of a *property redefinition* must be compatible with the original *property definition* in the *generalization*;
- and the type of the arguments in a *invocation* must be compatible with the type of the declared parameters of the corresponding *function*.

In order to allow the compiler to verify the types during the model validation, the model elements that must have an associated type are all specializations of the *TypedElement* metaclass, which has the abstract property *type*. Each *TypedElement* specialization must redefine the *type* property in order to be able to infer its type.

All types are specializations of the metaclass *Type*, which represents the *type declaration* or *type inference* of a *TypedElement*.

Among other properties, *Type* defines the *min_cardinality* and the *max_cardinality* allowed on a *TypedElement*. CML only allows specific cardinality options:

- A *type* with the min/max cardinality equal to one is called "required". It is declared as required when no suffix is provided in a *type declaration*.
- A *type* with cardinality zero-or-one is called "optional". It is declared as optional by the question-mark (?) suffix in a *type declaration*.
- A *type* with cardinality zero-or-more is called "sequential". It is declared as sequential by the asterisk-mark (*) suffix in a *type declaration*.

The direct specializations of *Type* are the following metaclasses:

- *ValueType*: for *types* that contain *values*, such as the *primitive types* (§ 31).
- *ReferenceType*: for *types* that contain *references* to the actual *instances*, such as the *concept types* (§ 36).
- *TupleType*: for types that declare *tuples* used as an *argument* or a *result* of a *function* (§ 26).
- *FunctionType*: used in *functions* to declare that an *argument* accepts a *lambda expression* (§ 24).

The instances of the *Type* specializations listed above are presented in the following chapters.

Thirty-one

Primitive Types

A *primitive type* in CML is one of the pre-defined *value types* supported by the language, as shown in tables 31.1 and 31.2.

In the ER [1] metamodel, a *data type* is formally defined as a *set* of *values* that can be held by an *attribute* (§ 6). The original ER paper [1] states that, for each *value set* (i.e. *data type*), there is a *predicate* that can be used to test whether a *value* belongs to the *set*. In CML, instead, *literal expressions* are syntactically defined for each *primitive type*, so that the *type* can be inferred from the *literal expression*.

On the original ER paper, it is also said that *values* in a *value set* may be equivalent to *values* in another *value set*. In CML, also, *literal expressions* of the *Integer* type may be equivalent to *literal expressions* of the *Decimal* type, and so with other *numeric types*. This allows *expressions* (§ 12) of a *primitive type* to be promoted to *expressions* of another *primitive type* in order to allow *type inference* of composite *expressions*.

In the UML [4] metamodel, there is a specific metaclass named *PrimitiveType*, which matches to the same notion in CML.

31.1 Example

Figure 31.1 presents examples of *atributes* declared with *primitive types* in CML. Each example corresponds to one of the *primitive types* supported by the language, as shown in tables 31.1 and 31.2. The *target constructors* (§ 40) of CML's base module will translate the primitive types to Java, C#, C/C++, Python, and TypeScript (JavaScript), according to the mapping shown in the tables.

```
@concept PrimitiveTypes
{
    -- Core Primitive Types:

    -- Only values are the literal expressions: true, false
    a: Boolean;

    -- 32-bit signed two's complement integer
    c: Integer;

    -- Arbitrary precision arithmetic.
    -- BigDecimal in Java; decimal in C#; decimal128 in C++.
    d: Decimal;

    -- 16-bit Unicode character sequences
    -- as in Java, C#, C++ (std::wstring), and JavaScript.
    b: String;

    -- Additional Primitive Types:

    -- 8-bit signed two's complement integer
    e: Byte;

    -- 16-bit signed two's complement integer
    f: Short;

    -- 64-bit signed two's complement integer
    g: Long;

    -- 32-bit IEEE 754 floating point
    h: Float;

    -- 64-bit IEEE 754 floating point
    i: Double;
}
```

Figure 31.1: Example of *Primitive Types*

CML	Java	C#	C++	Python	TypeScript (JavaScript)
String	String	string	std::wstring	str	string
16-bit Unicode character sequences.					
Boolean	boolean	bool	bool	bool	boolean
Only values are the literal expressions: **true**, **false**.					
Integer	int	int	int32_t	int	number
32-bit signed two's complement integer.					
Decimal*	BigDecimal	decimal	decimal128	Decimal	number
Arbitrary precision, fixed-point, or decimal floating-point, depending on the target language.					

*The specification of Decimal type varies by target programming language. Compared to the binary floating-point types (Float and Double), the Decimal type is better suited for monetary calculations at a performance cost.

Table 31.1: Core Primitive Types in CML.

31.2 Syntax

Figure 31.2 specifies the syntax used to declare any kind of *type*, including *primitive types*. The NAME of the *type* may be any of the *primitive types* defined in the column named *CML* of the tables 31.1 and 31.2. Optionally, cardinality may also be specified for a *primitive type*. The '*' cardinality suffix allows zero or more values to be stored in a property as a collection type (**??**). The '?' cardinality suffix allows a single value to be stored, or none. If no cardinality is specified, a value must be assigned to the *attribute* when its *concept* is instantiated.

Figure **??** presents the *Type* metaclass in an EMOF [5] class diagram of the CML metamodel, and figure 31.3 specifies the transformation from the *type* concrete syntax to its abstract syntax.

CML	Java	C#	C++	Python	TypeScript (JavaScript)	Specification
Byte	byte	byte	int8_t	int	number	8-bit signed two's complement integer
Short	short	short	int16_t	int	number	16-bit signed two's complement integer
Long	long	long	int64_t	long	number	64-bit signed two's complement integer
Float	float	float	float*	float	number	32-bit IEEE 754 binary floating point
Double	double	double	double*	float	number	64-bit IEEE 754 binary floating point

*C++ floating point types may vary by hardware and compiler

Table 31.2: Additional Primitive Types in CML.

```
typeDeclaration returns [Type type]:
    NAME cardinality?;

cardinality:
    ('?' | '*');
```

Figure 31.2: Type Declaration Syntax

```
node Type: NAME CARDINALITY?
{
    name = NAME;
    cardinality = CARDINALITY?;
}
```

Figure 31.3: Type AST Instantiation

Thirty-two

Boolean Type

The *Boolean* type accepts only two literal expressions (§ 13): `true` or `false`. When either of these two literals is found in *expressions* (§ 12), its *type* is inferred to be *Boolean*.

Any *expression* of the *Boolean* type may be used as an operand of *logical expressions* (§ 15).

Thirty-three

Numeric Types

The *numeric types* are the following in CML (from the smallest set to the largest one): *Byte*, *Short*, *Integer*, *Long* and *Decimal*. They belong to the group of *primitive types* (§ 31), which are shown in tables 31.1 and 31.2.

There is a specific *literal expression syntax* (§ 13) for each *numeric type*, which allows the type to be inferred uniquely. *Literal expressions* of a narrower-range type may be equivalent to literal expressions of the wider-range one. This allows type inference of *arithmetic expressions* (§ 14) and *relational expressions* (§ 16).

Thirty-four

Floating-Point Types

The *floating-point types* in CML are *Float* and *Double* for single-precision (32-bit) and double-precision (64-bit), respectively. They belong to the group of *primitive types* (§ 31), which are shown in tables 31.1 and 31.2.

Each *floating-point type* has a specific *literal expression syntax* (§ 13), which allows them to be inferred uniquely. *Literal expressions* of the *Float* type may be equivalent to *literal expressions* within the same range of the *Double* type. This allows type inference of *arithmetic expressions* (§ 14) and *relational expressions* (§ 16).

In order to prevent data/precision loss, it is disallowed the coercion of a value of a *numeric type* (§ 33) to a value of a *floating-point type*, and vice-versa. That means it is only possible to combine *expressions* (§ 12) of *floating-point* and *numeric* types with explicit conversion.

Thirty-five

String Type

The *String* type in CML represents a 16-bit Unicode character sequence, matching the same type in other programming languages, as described in table 31.1.

There is a specific *literal expression syntax* (§ 13) for *String* values, which allows them to be inferred uniquely. *Expressions* (§ 12) of the *String* type may be the operands of *relational expressions* (§ 16). They may also be combined with the ampersand (*&*) operator for concatenation (§ 21), in which case the resulting *expression* is of *String* type.

Thirty-six

Reference Types

In CML, all *type declarations* referring to the name of a *concept* (§ 4) are instances of the *ReferenceType* metaclass (§ 30); in short, the *type declaration* declares a *reference type*. A *property* (§ 5) of a concept A, whose type is declared or inferred to be of a concept B, holds a reference to an instance of concept B; not the actual instance. This allows the *properties* of a concept C to also reference the same instance of B.

Models in CML do not to keep track of the memory used to store the actual instances. CML expects the target programming language or technology to support some kind of reference management, such as a garbage collector in Java or automatic reference counting in Swift, or still a database. CML does not require any particular implementation.

The *path expressions* (§ 22) whose result is of a *reference type* may be used in *referential expressions* (§ 17), *type-checking expressions* (**??**), *type-casting expressions* (**??**) and in *invocations* (§ 23).

Thirty-seven

Tuple Types

Tuples may be declared as the *type* of an *argument* or the *result* of a *function* (§ 26). A *type declaration* that declares a *tuple* is an instance of the *TupleType* metaclass.

Thirty-eight

Function Types

A *function type declaration* may used in *functions* (§ 26) to declare that an *argument* accepts a *lambda expression* (§ 24).

Part VI

Code Generation

Thirty-nine

Templates

Forty

Constructors

Forty-one

Tasks

Forty-two

Targets

Part VII

Organization and Sharing

Forty-three

Modules

Forty-four

Libraries

Part VIII

Appendices

A

CML Concrete Syntax (Grammar)

A.1 ANTLR Grammar

```
// Compilation Units:

compilationUnit:
    declarations*;

declarations:
    moduleDeclaration | conceptDeclaration | associationDeclaration | taskDeclaratio


// Concept Declarations:

conceptDeclaration returns [Concept concept]:
    ABSTRACT? 'concept' NAME
    (':' generalizations)?
    (';' | propertyList);

generalizations:
    NAME (',' NAME)*;



// Property Declarations:

propertyList:
    '{' (propertyDeclaration ';')* '}';

propertyDeclaration returns [Property property]:
   DERIVED? NAME (':' typeDeclaration)? ('=' expression)?;

DERIVED: '/';

// Type Declarations:

typeDeclaration returns [Type type]:
    NAME cardinality?;

cardinality:
    ('?' | '*');
```

```
// Target Declarations:

targetDeclaration returns [Target target]:
    'target' NAME propertyList;



// Names:

// All keywords must be declared before NAME.
// Otherwise, they are recognized as a NAME instead.

FOR: 'for';
IN: 'in';

SELECT: 'select';
REJECT: 'reject';

YIELD: 'yield';
RECURSE: 'recurse';

INCLUDES: 'includes';
EXCLUDES: 'excludes';

EVERY: 'every';
EXISTS: 'exists';

FROM: 'from';
REDUCE: 'reduce';

TAKE: 'take';
DROP: 'drop';

FIRST: 'first';
LAST: 'last';

COUNT: 'count';
SUM: 'sum';
AVERAGE: 'average';
MAX: 'max';
MIN: 'min';
REVERSE: 'reverse';
```

```
UNIQUE: 'unique';
WHILE: 'while';


IF: 'if';

THEN: 'then';

ELSE: 'else';

BOOLEAN: 'true' | 'false';

AND: 'and';

OR: 'or';

XOR: 'xor';

IMPLIES: 'implies';

NOT: 'not';

ABSTRACT:
    'abstract';

NAME:
    ('A'..'Z' | 'a'..'z')
    ( 'A'..'Z' | 'a'..'z' | '0'..'9' | '_' )*;


// Literals:

literalExpression returns [Literal literal]: BOOLEAN | STRING | INTEGER | LONG | SHO

STRING:
    '"' (ESC | . )*? '"';

fragment ESC: '\\'[btnr"\\];

INTEGER:
    ('0'..'9')+;
```

```
LONG:
    ('0'..'9')+ 'l';

SHORT:
    ('0'..'9')+ 's';

BYTE:
    ('0'..'9')+ 'b';

DECIMAL:
    ('0'..'9')* '.' ('0'..'9')+;

FLOAT:
    ('0'..'9')* '.' ('0'..'9')+ 'f';

DOUBLE:
    ('0'..'9')* '.' ('0'..'9')+ 'd';

// Ignoring Whitespace:

WS:
    ( ' ' | '\t' | '\f' | '\n' | '\r' )+ -> skip;


// Ignoring Comments:

COMMENT:
    (('//' | '-') .*? '\n' | '/*' .*? '*/' ) -> skip;
```

B

CML Abstract Syntax (Metamodel)

C

CML Abstract Syntax Tree (Instantiation)

D

CML Constraints (Validations)

```
context Concept

inv unique_concept_name:
    parent.concepts
        ->select(c| c != self and c.name = self.name)
        ->isEmpty()

context Property
inv unique_property_name:
    self.scope.properties
        ->select(p| p != self and p.name = self.name)
        ->isEmpty()

context Property
inv property_type_specified_or_inferred:
    type->notEmpty() or expression->notEmpty()

context Property
inv property_type_assignable_from_expression_type:
    type->notEmpty() and expression->notEmpty() implies
        type.isAssignableFrom(expression.type)
```

E

Language Specification Notation

Bibliography

[1] Peter Pin-Shan Chen. The Entity-Relationship Model (Reprinted Historic Data). In David W. Embley and Bernhard Thalheim, editors, *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, pages 57–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[2] Torben ï£¡gidius Mogensen. *Introduction to Compiler Design.* Undergraduate Topics in Computer Science. Springer, 2011.

[3] OMG. Object Constraint Language (OCL), Version 2.4, 2014.

[4] OMG. Unified Modeling Language (UML), Superstructure, Version 2.5, 2015.

[5] OMG. Meta Object Facility (MOF) Core Specification, Version 2.5.1, 2016.

[6] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[7] Terence John Parr. Enforcing Strict Model-view Separation in Template Engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233. ACM, New York, NY, USA, 2004.