Conceptual Modeling Language Specification

Version 1.0 (Draft)

Quenio Cesar Machado dos Santos

Universidade Federal de Santa Catarina*

July 2017

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

Contents

1 Introduction **1**
1.1 The CML Compiler 1
1.2 Organization and Notations 3

2 Concepts **5**
2.1 Properties 8
2.2 Generalization / Specialization 10
2.3 Abstract Concepts 13

3 Attributes **18**
3.1 Primitive Types 19
3.2 Derived Attributes 23

4 Associations **26**
4.1 Unidirectional Associations 26
4.2 Bidirectional Associations 26
4.3 Collection Types 26
4.4 Collection Types 26
4.5 Derived Associations 26

5 Expressions **27**
5.1 Literal Values 28
5.2 Prefix Expressions 28
5.3 Infix Expressions 28
5.4 Conditional Expressions 28
5.5 Path Expressions 28

6 Queries **32**

7 Code Generation **33**
7.1 Tasks 33
7.2 Constructors 33
7.3 Templates 33
7.4 Targets 33

8 Modules **34**

9 Libraries **35**

A CML Concrete Syntax (Grammar) **36**
A.1 ANTLR Grammar 37

B CML Abstract Syntax (Metamodel) **41**

C CML Abstract Syntax Tree (Instantiation) **42**

D CML Constraints (Validations) **43**

E Language Specification Notation **45**

Bibliography **46**

List of Figures

1.1 An architectural overview of the CML compiler. 2

2.1 Concept Examples 6
2.2 Concept Declaration Syntax 6
2.3 Concept Metamodel 7
2.4 Concept AST Instantiation 7
2.5 Concept Constraints 8
2.6 Property Examples 8
2.7 Property Declaration Syntax 9
2.8 Property AST Instantiation 9
2.9 Property Constraints 10
2.10 Generalization Examples 12
2.11 Generalization Constraints 14
2.12 Abstract Concept Example 15
2.13 Abstract Concept Constraints 17

3.1 Examples of Attributes 19
3.2 Example of *Primitive Types* 21
3.3 Type Declaration Syntax 21
3.4 Type AST Instantiation 22
3.5 Auxiliary Methods of The *Type* Metaclass 24
3.6 The *isAssignableFrom()* Method of The *Type* Metaclass 25

5.1 Expression Examples 28
5.2 Expressions Syntax 29
5.3 Expression Metamodel 30
5.4 Literals Lexical Structure 31

List of Tables

3.1 Core Primitive Types in CML. 20
3.2 Additional Primitive Types in CML. 22

One

Introduction

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.

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.

Section § 1.1 will provide an overview of the CML compiler's architecture. Section § 1.2 describes the organization and notation used in the remainder of this document.

1.1 The CML Compiler

The CML compiler's overall architecture follows the standard compiler design literature [2]. An overview diagram of the architecture is shown in figure 1.1.

The two main components of the compiler, and the artifacts they work with, are presented in the next subsections.

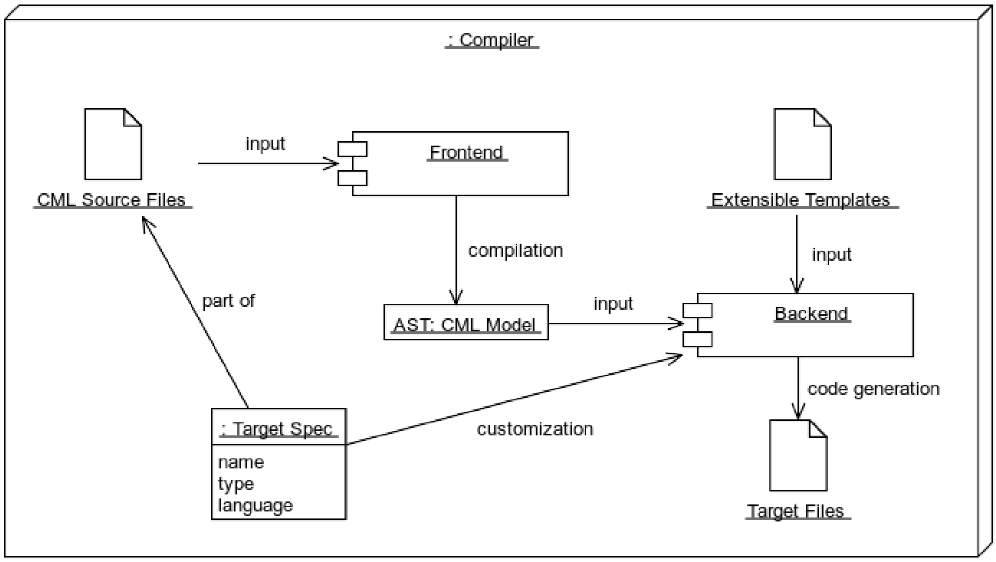

Figure 1.1: An architectural overview of the CML compiler.

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 for the *backend* component.

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 *target specification* plays a key role in order to determine 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 *target specification* (specified in chapter §7.4) 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. The type of model element received as input by the *content template* depends on which function of the *files template* has defined the file to be generated.

1.2 Organization and Notations

The following chapters will specify every element of CML metamodel. Each chapter starts with a definition, followed by: an example; the specification of the concrete syntax; and then presenting the abstract syntax, and how to transform the concrete syntax into the abstract one.

Chapters may also have sections that specify sub-elements of the top-level CML metamodel element being described in the chapter level. Each sub-element is described under its section using the same definition structure (detailed below) that is used to define the top-level elements.

Definition. The definition of each CML metamodel element is stated in plain English on a paraprah (such as this one) starting with the "**Definition.**" heading. 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.

Examples. For each metamodel element declaration in CML, examples are provided on a paraprah (such as this one), starting with the "**Examples.**" heading. This type of paragraph refers 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.

Concrete Syntax. The concrete syntax of each CML metamodel element is described on a paragraph (such as this one), starting with the "**Concrete Syntax.**" heading. This type of paragraph 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.

Abstract Syntax. The abstract syntax of each CML metamodel element is described on a paragraph (such as this one), starting with the "**Abstract Syntax.**" heading. This type of paragraph 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 section § 1.1). The notation used to specify the transformations is presented in the appendix § E. Both figures must be considered normative.

Constraints. The constraints of each CML metamodel element are described on a paragraph (such as this one), starting with the "**Constraints.**" heading. This type of paragraph refers 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 CML metamodel element, the corresponding chapter or section defining the other element is provided in parenthesis, like so (§ 1.2).

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.

Two

Concepts

Definition. 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 type*; in UML [4], to a *class*. The CML *concept* differs, however, from the UML *class*, because it has only *properties* (§ 2.1), while the UML *class* may also have *operations*.

Examples. Figure 2.1 presents some examples of *concepts* declared in CML. As shown, a *concept* may have zero or more *properties* (§ 2.1), and a *property* may optionally declare a *type* (§ 3.1, § 4.3). Also, as shown in the concept **EBook** of the example, a *concept* may specialize (§ 2.2) another *concept*.

Concrete Syntax. Figure 2.2 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 (§ 2.2) of the declared *concept*. A list of *properties* (§ 2.1) may be declared under the **concept** block. And the **abstract** keyword may precede the **concept** keyword, making a *concept* abstract (§ 2.3).

Abstract Syntax. Figure 2.3 presents the *concept* metamodel in an EMOF [5] class diagram, and figure 2.4 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:

- *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*.

```
// 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;
```

Figure 2.1: Concept Examples

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

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

Figure 2.2: Concept Declaration Syntax

- *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*.

Constraints. Figure 2.5 presents the invariants of the *concept* metamodel:

- *unique_concept_name*: Each *concept* must have a unique NAME within its *module* (§ 8).

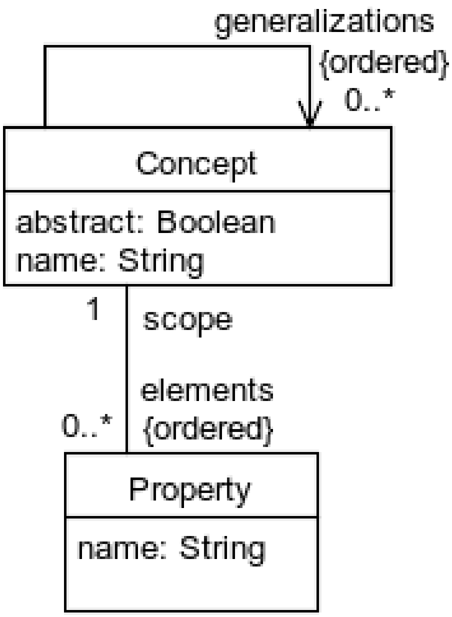

Figure 2.3: Concept Metamodel

```
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)*;
```

Figure 2.4: Concept AST Instantiation

```
context Concept

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

Figure 2.5: Concept Constraints

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

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

Figure 2.6: Property Examples

2.1 Properties

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

Examples. Figure 2.6 presents some examples of *properties* declared in CML. As shown in the examples, a *property* may be an *attribute* (§ 3) of a *primitive type* (§ 3.1), or represent the role/end of an *association* (§ 4).

Concrete Syntax. Figure 2.7 specifies the syntax used to declare a *property*. The NAME is followed by a *typeDeclaration* (§ 3.1 and § 4.3). Optionally, an *expression* (§ 5) may be specified in order to set the initial value.

Abstract Syntax. Figure **??** presents the *Property* metaclass in an EMOF [5] class diagram of the CML metamodel, and figure 2.8 specifies the transforma-

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

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

DERIVED: '/';
```

Figure 2.7: Property Declaration Syntax

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

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

Figure 2.8: Property AST Instantiation

tion 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*.

Constraints. Figure 2.9 presents the invariants of the *Property* metaclass:

- *unique_property_name*: Each *property* must have a unique NAME within its *concept* (§ 2).
- *property_type_specified_or_inferred*: Either the *property* explicitly defines a *type* or it defines an *expression*, from which the type is inferred.

```
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)
```

Figure 2.9: Property Constraints

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).

2.2 Generalization / Specialization

Definition. A *concept* (§ 2) 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* (§ 2.1) 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*.

Examples. Figure 2.10 presents some examples of generalization/specialization relationships declared in CML. As shown, a *concept* (§ 2) 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* (§ 3), such as **color** and **area**, or also the *roles* in *unidirecional associations* (§ 4.1). Both *attributes* and *roles* are *properties* (§ 2.1) 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.

Concrete Syntax. Figure 2.2 specifies the syntax used to declare a *concept* (§ 2), 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.

Abstract Syntax. Figure 2.3 presents the *Concept* metaclass in an EMOF [5] class diagram of the CML metamodel, and figure 2.4 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*.

Constraints. Figure 2.11 presents the invariants of the *Concept* and *Property* classes related to *generalizations*:

- *not_own_generalization*: A *concept* (§ 2) 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*.

```
-- 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;
}

-- 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;
}
```

Figure 2.10: Generalization Examples

- *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.

2.3 Abstract Concepts

Definition. An *abstract concept* is one that does not represent specific instances, but instead serves as a *generalization* (§ 2.2) for other *concepts*, which in turn represent specific instances. Thus, all instances of an *abstract concept* are first instances of its *specializations*. CML supports tagging a *concept* as *abstract*. An *abstract concept* in CML may also define a *derived property* (§ 2.1) wihtout providing an *expression* (§ 5) in its definition; such *properties* may also be called *abstract properties*. CML's support for *abstract concepts* matches UML's [4], which allows the declaration of *abstract classes* – by setting the *isAbstract* attribute of the *Class* metaclass instance to *true*. UML also allows the declaration of corresponding *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*.

Examples. Figure 2.12 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* (§ 2.1) without an *expression* (§ 5). 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**.

Concrete Syntax. Figure 2.2 specifies the syntax used to declare a *concept* (§ 2) in CML. It shows that a *concept* may be tagged with the **abstract** keyword

```
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
```

```
-- As an abstract concept,
-- no direct instances of Shape are ever created.
abstract concept 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;
}
```

Figure 2.12: Abstract Concept Example

in order to convey it as an *abstract concept*. Figure 2.7 specifies the syntax used to declare a *property* (§ 2.1) 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*.

Abstract Syntax. Figure 2.3 presents the *concept* metamodel in an EMOF [5] class diagram, and figure 2.4 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.

Constraints. Figure 2.13 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*.

```
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
```

Figure 2.13: Abstract Concept Constraints

Three

Attributes

Definition. In CML, *attributes* are *properties* (§ 2.1) of *primitive types* (§ 3.1). 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* (§ 5). Some *attributes*, however, may be constantly derive their *value* from an *expression* (not only initially), in which case they are called *derived attributes* (§ 3.2). While initial values are only set when a *concept* (§ 2) is instantiated, the value of *derived attributes* is always evaluated from the given *expression*, and they cannot be set any other way.

Examples. Figure 3.1 presents some examples of *attributes* declared in CML. As shown, the attribute **a** is a regular attribute definition that specifies the *primitive type* (§ 3.1) 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*.

Concrete Syntax. Figure 2.7 specifies the syntax used to declare any kind of *property* (§ 2.1), including *attributes*. The NAME of an *attribute* is followed by a *typeDeclaration* of a *primitive type* (§ 3.1). Optionally, an *expression* (§ 5) 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.

```
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"
}
```

Figure 3.1: Examples of Attributes

Abstract Syntax. Since an *attribute* in CML is just a *property* (§ 2.1) with *primitive types* (§ 3.1), 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 2.8 specifies the *property* transformation from its concrete syntax to its abstract syntax.

3.1 Primitive Types

Definition. A *primitive type* in CML is one of the pre-defined *data types* supported by the language, as shown in tables 3.1 and 3.2. In the ER [1] metamodel, a *data type* is formally defined as a *set* of *values* that can be held by an *attribute* (§ 3). 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*, and so with other *ordinal types*. This allows *expressions* of a *primitive type* to be promoted to *expressions* of another *primitive type* in order to allow *type inference* of composite *expressions*, such as *infix expressions* (§ 5.3). In the UML [4] metamodel, there is a specific metaclass named *PrimitiveType*, which matches to the same notion in CML.

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 3.1: Core Primitive Types in CML.

Examples. Figure 3.2 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 3.1 and 3.2. The *target constructors* (§ 7.2) 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.

Concrete Syntax. Figure 3.3 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 3.1 and 3.2. 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 (§ 4.3). 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.

```
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 3.2: Example of *Primitive Types*

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

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

Figure 3.3: Type Declaration 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 3.2: Additional Primitive Types in CML.

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

Figure 3.4: Type AST Instantiation

Abstract Syntax. Figure **??** presents the *Type* metaclass in an EMOF [5] class diagram of the CML metamodel, and figure 3.4 specifies the transformation from the *type* concrete syntax to its abstract syntax.

Constraints. Figures 3.5 and 3.6 define the *isAssignableFrom()* operation in the *Type* metaclass, which is used by the *property_type_assignable_from_expression_type* constraint in figure 2.9. Basically, one of the following conditions must be met for a source *type* to be assignable to a destination *type*:

- The source *type* has the same name as the destination *type*.
- Both types are *numeric* and the destination *type* is wider than the source *type*. Caveat: Floating-point types (Float and Double) are never assignable to the other *numeric types* (Byte, Short, Integer, Long), and vice-versa.
- Both types refer to *concepts* and the destination *concept* is *generalization* (§ 2.2) of the source *concept*.

Additionally, one of the following conditions must be met regarding the *type*'s *cardinality*:

- The cardinality of the source *type* matches the cardinality of the destination *type*.
- The destination *type* has the *zero-or-one* cardinality and the source *type* has the *one* cardinality.
- The destination *type* has the *zero-or-more* cardinality and the source *type* has any other cardinality.

3.2 Derived Attributes

```
context Type::numeric: Boolean
def:
    types = Set {
        'Byte' 'Short' 'Integer' 'Long' 'Decimal'
    }
derive:
    types->includes(self.name)

context Type::isNumericWiderThan(Type other): Boolean
def:
    types = Sequence {
        'Byte' 'Short' 'Integer' 'Long' 'Decimal'
    }
pre:
    self.numeric and other.numeric
post:
    result =
        types->indexOf(self.name) > types->indexOf(other.name)

context Type::floating: Boolean
def:
    types = Set {
        'Float' 'Double'
    }
derive:
    types->includes(self.name)

context Type::isFloatingWiderThan(Type other): Boolean
def:
    types = Sequence {
        'Float' 'Double'
    }
pre:
    self.floating and other.floating
post:
    result =
        types->indexOf(self.name) > types->indexOf(other.name)
```

Figure 3.5: Auxiliary Methods of The *Type* Metaclass

```
context Type::isTypeAssignableFrom(Type other): Boolean
post:
    if self.name = other.name then
        result = true
    else if self.numeric and other.numeric then
        result = self.isNumericWiderThan(other)
    else if self.floating and other.floating then
        result = self.isFloatingWiderThan(other)
    else if self.concept->notEmpty() and
            other.concept->notEmpty()
         then
            result = other.concept.all_generalizations
                        ->exists(name = self.concept.name)
    else
        result = false

context Type::isCardinalityAssignableFrom(Type other): Boolean
post:
    result = (self.cardinality = other.cardinality)
          or (self.cardinality = '?' and other.cardinality = '')
          or (self.cardinality = '*')

context Type::isAssignableFrom(Type other): Boolean
post:
    result = self.isTypeAssignableFrom(other) and
             self.isCardinalityAssignableFrom(other)
```

Figure 3.6: The *isAssignableFrom()* Method of The *Type* Metaclass

Four

Associations

4.1 Unidirectional Associations

4.2 Bidirectional Associations

4.3 Collection Types

4.4 Collection Types

4.5 Derived Associations

Five

Expressions

Definition. An *expression* in CML is used to compute values and collections that initialize *properties* or define *derived properties*. On the UML [4] metamodel, it corresponds to an *Expression*; in OCL [3], to *OclExpressionCS*. The CML *expressions* are designed to provide the same level of expressivity provided by OCL *expressions*, but the CML syntax varies from OCL, especially for collection operations.

Examples. Figure 5.1 has some examples of CML *expressions*. As shown, there are different types of expressions: literals (§ 5.1), prefix expressions (§ 5.2), infix expressions (§ 5.3), conditional expressions (§ 5.4), path expressions (§ 5.5) and queries (§ 6).

Concrete Syntax. Figure 5.2 specifies the syntax of all CML *expressions*. It also lists them in their order of precedence. Observe that the grammar in figure 5.2 has left recursions, and thus is ambiguous. However, ANTLR [6] will use the order in which the alternatives are listed in order to resolve the ambiguity, and so define the precedence among the operators. 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.

Abstract Syntax. Figure 5.3 presents the *Expression* metamodel 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, for the *Literal* subclass, **kind = "literal"**.

```
concept Expressions
{
    // Literals:
    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:
    if_then_else = if a > 0 then a else b;

    // Path:
    path = somePath.bar;

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

Figure 5.1: Expression Examples

- *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.

5.1 Literal Values

5.2 Prefix Expressions

5.3 Infix Expressions

5.4 Conditional Expressions

5.5 Path Expressions

```
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?;
```

Figure 5.2: Expressions Syntax

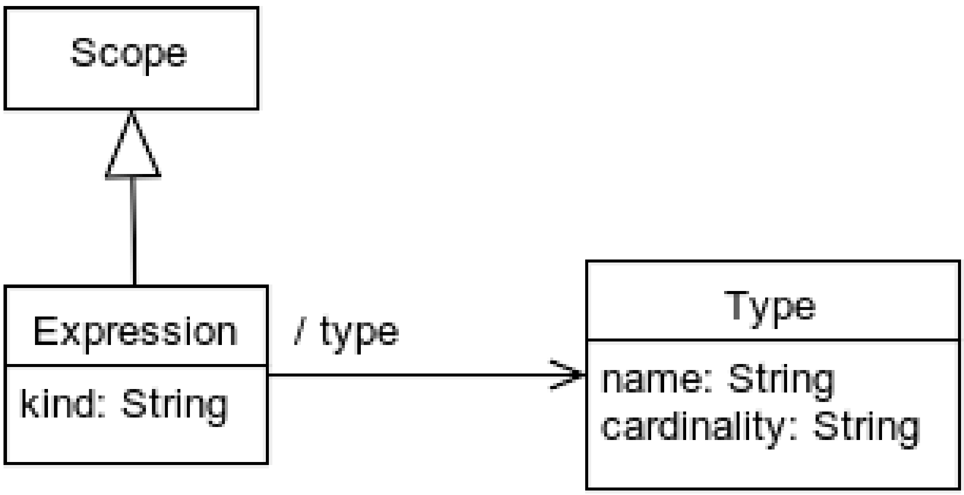

Figure 5.3: Expression Metamodel

```
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';
```

Figure 5.4: Literals Lexical Structure

Six

Queries

Seven

Code Generation

7.1 Tasks

7.2 Constructors

7.3 Templates

7.4 Targets

Eight

Modules

Nine

Libraries

A

CML Concrete Syntax (Grammar)

A.1 ANTLR Grammar

```
// Compilation Units:

compilationUnit:
    declarations*;

declarations:
    moduleDeclaration | conceptDeclaration | taskDeclaration;


// 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.