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

List of Figures

List of Tables

One

Introduction

The *Conceptual Modeling Language* (CML) is specified in this document. It allows modeling the information of software systems, focusing on the structural aspects. Using CML, it is possible to represent the information as understood by the system users, disregarding its physical organization as implemented by the 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.

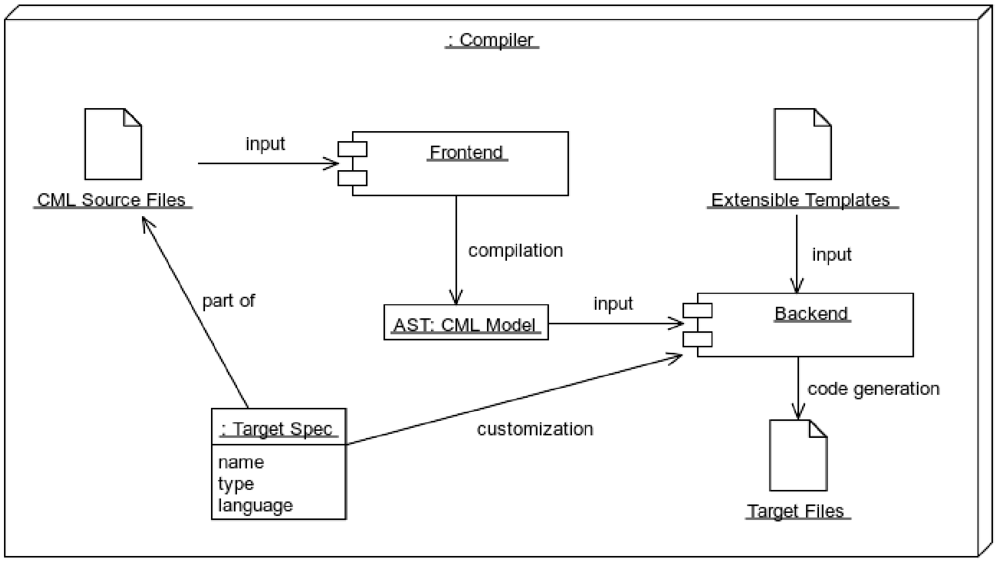

Figure 1.1: An architectural overview of the CML compiler.

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 **??**) 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. 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 meaning in a domain. On the ER [1] metamodel, it corresponds to an *entity*; 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 in the examples, 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 last example of figure 2.1, 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
    (':' ancestorList)?
    (';' | propertyList);

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

Figure 2.2: Concept Declaration Syntax

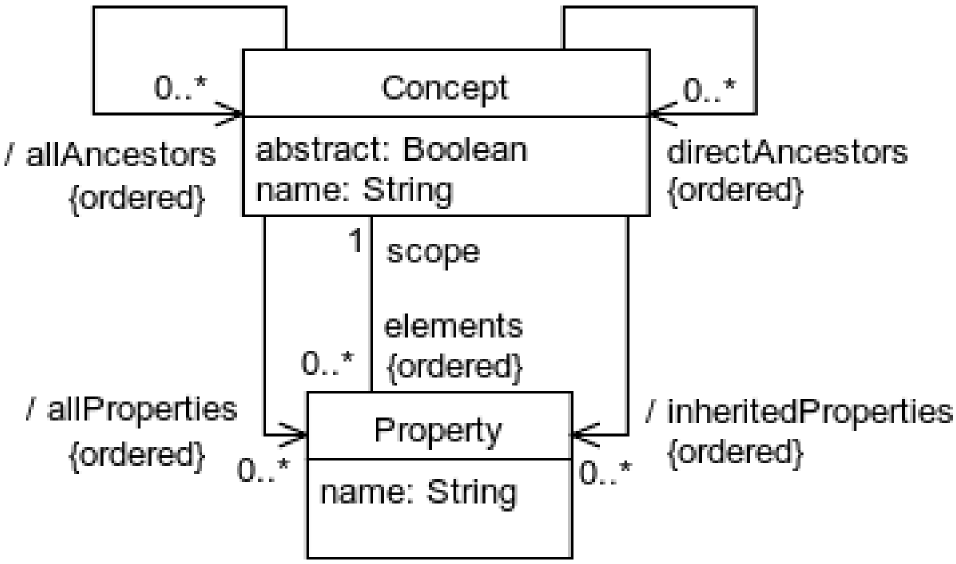

Figure 2.3: Concept Metamodel

- *elements*: an *ordered set* referencing all *properties* parsed in the **concept** block.
- *directAncestors*: an *ordered set* referencing all *concepts* whose NAMEs were enumerated in the *AncestorList*.

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).
- *not_own_ancestor*: A *concept* may not be listed on its own *AncestorList*, nor on the *AncestorList* of its direct or indirect ancestors (§ 2.2).

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.

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

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

Figure 2.4: Concept AST Instantiation

```
context Concept

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

inv not_own_ancestor:
    self.allAncestors
        ->select(c| c = self)
        ->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

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

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

Figure 2.7: Property Declaration Syntax

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* metamodel in an EMOF [5] class diagram, and figure 2.8 specifies the transformation from the *property* concrete syntax to its abstract syntax. For each *property* parsed by the compiler,

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

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

Figure 2.8: Property AST Instantiation

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*.

2.2 Generalization/Specialization

2.3 Abstract Concepts

Three

Attributes

3.1 Primitive Types

Four

Associations

4.1 Unidirectional Associations

4.2 Bidirectional Associations

4.3 Collection Types

Five

Expressions

Definition. An *expression* may be used in CML 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 presents 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 define the precedence between 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.

```
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_transform = items | select name == "this";
}
```

Figure 5.1: Expression Examples

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;

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

```
literalExpression returns [Literal literal]: BOOLEAN | STRING | INTEGER | DECIMAL;

STRING:
    '"' .*? '"';

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

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

Figure 5.3: 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
    (':' ancestorList)?
    (';' | propertyList);

ancestorList:
    NAME (',' NAME)*;



// Property Declarations:

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

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


// 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 | DECIMAL;

STRING:
    '"' .*? '"';

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

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

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

inv not_own_ancestor:
    self.allAncestors
        ->select(c| c = self)
        ->isEmpty()

context Property

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

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.