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 compilers 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.

Chapter 2 will provide an overview of the CML compiler's architecture. The following chapters will specify every element of CML metamodel. Each chapter starts with an example, followed by the specification of the concrete syntax, and then presenting the corresponding EMOF [5] class diagram with the abstract syntax.

Two

Compiler

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

2.1 Compiler 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

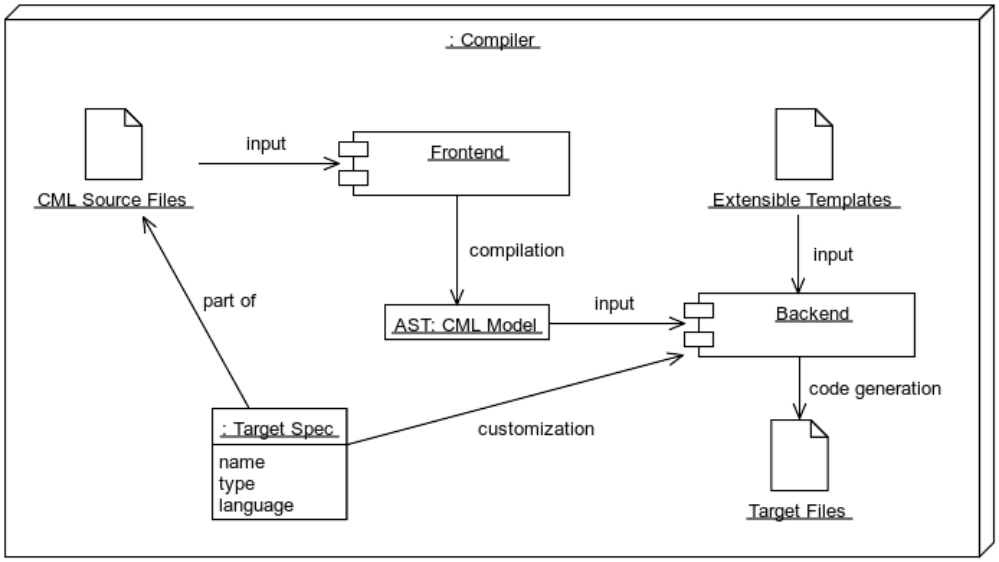

Figure 2.1: An architectural overview of the CML compiler.

will be executed at this point. Any errors are presented to the developer, interrupting the progress to the next phase. If the *source files* are parsed and validated successfully, then an internal representation (AST) of the *CML model* is generated. The AST serves then as the input for the *backend* component.

2.2 Compiler 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 [6]. 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) 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.

Three

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*, while the UML *class* may also have *operations*.

Examples. Figure 3.1 presents some examples of *concepts* declared in CML. As shown in the examples, a *concept* may have zero or more *properties* (§ 3.1), and a *property* may optionally declare a *type* (§ 4.2, § 5.3). Also, as shown in the last example of figure 3.1, a *concept* may specialize (§ 3.2) another *concept*.

Concrete Syntax. Figure 3.2 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 (§ 3.2) of the declared *concept*. A list of *properties* (§ 3.1) may be declared under the **concept** block. And the **abstract** keyword may precede the **concept** keyword, making a *concept* abstract (§ 3.3).

Abstract Syntax. Figure 3.3 presents the *concept* metamodel in an EMOF [5] class diagram, and figure 3.4 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 3.1: Concept Examples

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

ancestorList:
    NAME (',' NAME)*;

ABSTRACT:
    'abstract';
```

Figure 3.2: Concept Declaration Syntax

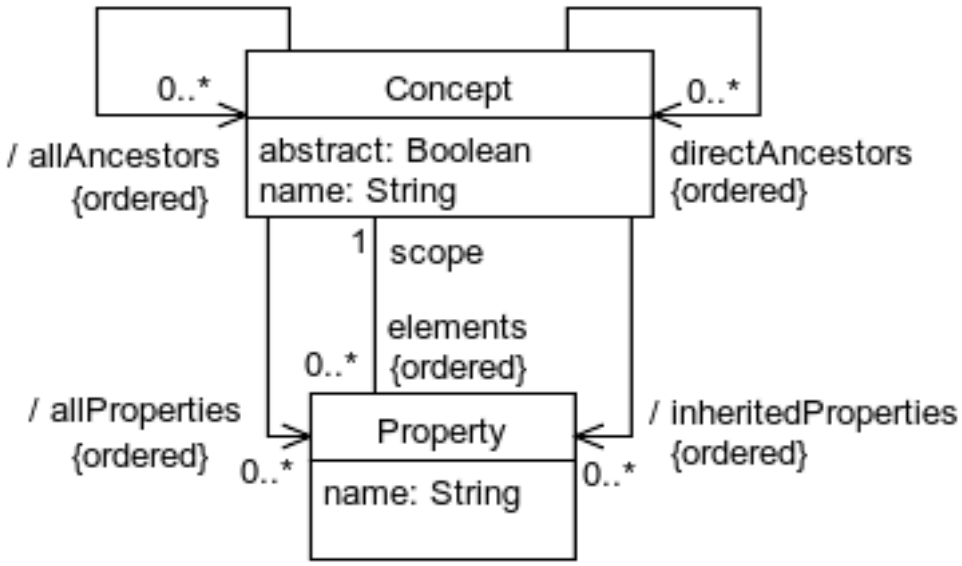

Figure 3.3: Concept Metamodel

- *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*.

3.1 Properties

3.2 Generalization/Specialization

3.3 Abstract Concepts

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

Figure 3.4: Concept AST Instantiation

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

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

Figure 3.5: Properties Declaration Syntax

Four

Attributes

4.1 Literal Values

4.2 Primitive Types

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

Figure 4.1: Literals Lexical Structure

Five

Associations

5.1 Unidirectional Associations

5.2 Bidirectional Associations

5.3 Collection Types

Six

Expressions

Seven

Targets

Eight

Modules and Libraries

A

Concrete Syntax (Grammar)

A.1 ANTLR Grammar

```
// Compilation Units:

compilationUnit returns [Model model]:
    declarations*;

declarations:
    conceptDeclaration | targetDeclaration;


// Concept Declarations:

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

ancestorList:
    NAME (',' NAME)*;

ABSTRACT:
    'abstract';


// Property Declarations:

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

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


// Type Declarations:

typeDeclaration returns [Type type]:
    NAME CARDINALITY?;

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

```
// Target Declarations:

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



// Names:

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


// Literals:

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


// Ignoring Whitespace:

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


// Ignoring Comments:

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

B

Abstract Syntax (Metamodel)

C

Abstract Syntax Tree (Instantiation)

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 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.