Conceptual Modeling Language Specification

Version 1.0

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 [**?**] and OCL [**?**];
- 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 [**?**] class diagram with the abstract syntax.

Two

Compiler Overview

The CML compiler's overall architecture follows the standard compiler design literature [**?**]. An overview diagram of the architecture is shown in figure 2.1.

The two main components of the compiler, and the artifacts they work

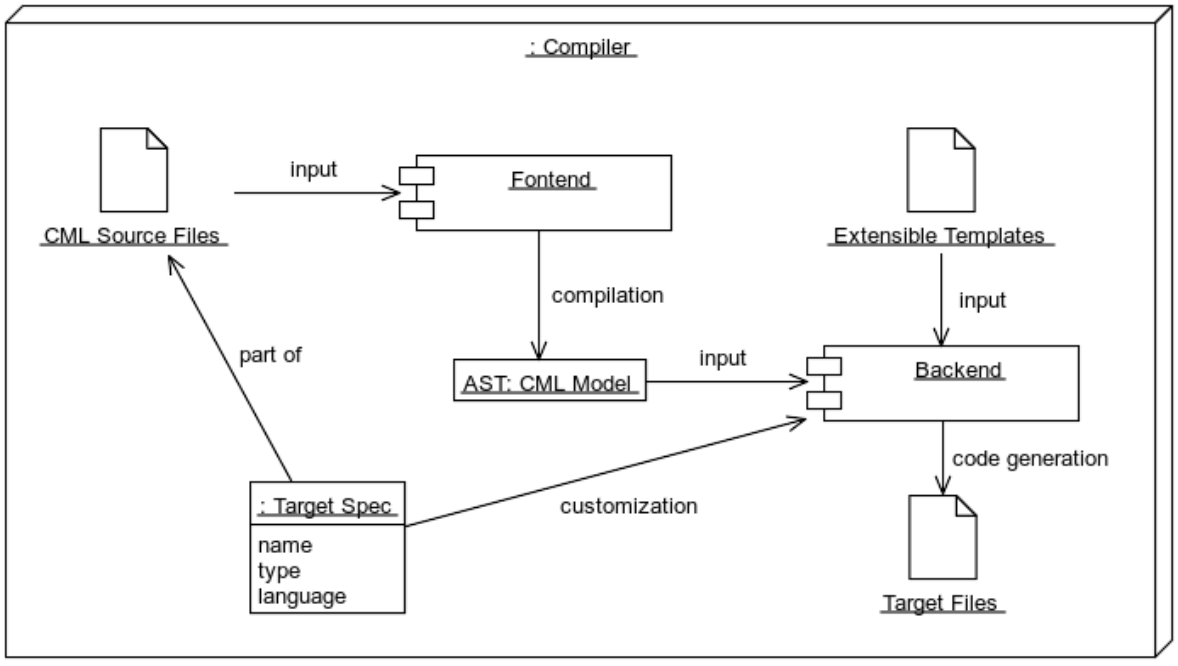

Figure 2.1: An architectural overview of the CML compiler.

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 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.

Three

Concepts

3.1 Properties

3.2 Inheritance

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

ancestorList:
    NAME (',' NAME)*;

ABSTRACT:
    'abstract';
```

Figure 3.1: Concept Declaration Syntax

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

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

Figure 3.2: Properties Declaration Syntax

Four

Associations

4.1 Unidirectional Associations

4.2 Bidirectional Associations

4.3 Collection Types

Five

Values

5.1 Literals

5.2 Primitive Types

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

Figure 5.1: Literals Lexical Structure

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)