Reusing Conceptual Models: Language and Extensible Compiler

Quenio Cesar Machado dos Santos[1] and Raul Sidnei Wazlawick[2]

[1] Computer Sciences,
UFSC - Universidade Federal de Santa Catarina, Brazil,
queniodossantos@gmail.com
[2] Associate Professor of Computer Sciences Department,
UFSC - Universidade Federal de Santa Catarina, Brazil,
raul@inf.ufsc.br

Abstract. This paper proposes a textual programming language that enables conceptual modeling (similarly to UML classes/associations and OCL constraints) and a compiler that allows code generation (via extensible textual templates) to any target language or technology. Together, the language and the compiler make it feasible to specify (in a single high-level language) the information of ever-changing, increasingly distributed software systems. From this single source, the automated code generation keeps the implementations (across the different platforms and technologies) consistent with the specification. Also, as the technology landscape evolves, these textual models allow the recurring use of the investment made on their specification. Unlike other approaches, such as MDA and MPS, the built-in tooling support, along with the textual nature of this programming language and its extensible templates, facilitates the integration to the workflow of software developers.

Keywords: conceptual modeling, programming language, code generation, model-driven software development, model-driven engineering, metaprogramming, generative programming

1 Introduction

In order to address the challenges of the ever-changing, increasingly distributed technologies used on software systems, the Model-Driven Architecture (MDA [9]) initiative by the Object Management Group (OMG) has been promoting model-driven software development. In particular, MDA has guided the use of high-level models (created with OMG standards, such as UML [11], OCL [10] and MOF [12]) to derive software artifacts and implementations via automated transformations. As one of its value propositions, the MDA guide [9] advocates:

> "Automation reduces the time and cost of realizing a design, reduces the time and cost for changes and maintenance and produces results that ensure consistency across all of the derived artifacts. For example, manually producing all of the web service artifacts required to implement

> a set of processes and services for an organization is difficult and error-prone. Producing execution artifacts from a model is more reliable and faster."

MDA provides guidance and standards in order to realize this vision, but it leaves to software vendors the task of providing the tools that automate the process of generating the implementations from the models.

The key role played by tools has been demonstrated by Voelter [18] in his *Generic Tools, Specific Languages* approach for model-driven software development. Voelter [18] has used domain-specific languages (DSL's) with the Metaprogramming System (MPS) in order to generate the software artifacts. Unlike MDA, which is based on UML/MOF models, MPS allows the specification of models using domain-specific editors. MPS itself is a generic tool, but it enables the definition of the abstract syntax, the editors and the code generators for DSL's.

The conceptual modeling language and extensible compiler presented here are an alternative approach to MPS. While the latter is a fully integrated development environment based on domain-specific languages and their projectional editors [3], the former (hereby called CML) is a compiler that has:

- as *input*, source files defined using its own conceptual language, which provides an abstract syntax similar to (but less comprehensive than) a combination of UML [11] and OCL [10];
- 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 section 2 addresses the question of why providing yet another language for conceptual modeling. The next two sections present the conceptual model language (section 3) and the compiler with its extensible templates (section 4). We conclude in section 5, reiterating the objectives achieved by CML and exploring future possibilities to expand CML's reach.

2 Why A New Language?

Thalheim [16] has observed that the choice of a conceptual modeling language has to do with its purpose. He suggests that a language is just a *carrier* mapping some properties of the *origin* (the problem space) that can provide utility to its users.

In this context, the purpose of the CML language is being a tool that allows software developers to transform conceptual models into executable code of an extensible range of technologies. In order to achieve this purpose, a new language is designed with the high-level goals presented in the following subsections.

[3] As explained by Voelter [18], projectional editors in MPS do not rely on parsers. Instead, the abstract syntax tree (AST) of the DSL is modified directly. The projection engine renders the visual representation of the AST based on the DSL editor definition.

2.1 Developer Experience

CML follows the principle laid out on the *Conceptual-Model Programming* (CMP) manifesto [2]:

> "Conceptual modeling is programming. The conceptual-model instance is the code, i.e., instead of 'the code is the model,' the phrase now becomes 'the model is the code.' A conceptual-model compiler assures that program execution corresponds to the conceptual specification, thus making the conceptual-model instance directly executable."

Furthermore, CML is also intended to enable software developers to do *conceptual modeling* on the same workflow they are used to doing *programming*. CML strives to *not only be* the code (as advocated by CMP), but also *look like* code (syntactically speaking), pursuing compatibility with developers' mindset, toolset and workflow. By providing its own syntax based on existing programming languages, CML then promotes the *modeling-as-programming* approach.

The UML [11] notation, on the other hand, being graphical, is not suited for mainstream, textual programming. However, the *Human Usable Textual Notation* (HUTN) [15] is a textual syntax for MOF-based [12] metamodels, and as such, it can also be used for UML models. The syntax of the structural (static) elements of CML models is based on HUTN.

The OWL 2 standard, as another example, provides the Manchester [19] syntax, which is intended to be user-friendly. However, it does not resemble the syntax of commonly used, block-based, imperative programming languages, such as C [5] and its syntax-alike descendants. Manchester's syntax is also unlike the syntax of declarative, query languages, such as SQL [6].

Using CML, and its familiar syntax (as we shall demonstrate in the next sections), it is expected that developers can raise the abstraction level of their programs.

2.2 Language Evolution

CML is expected to evolve with its compiler, and the tooling around it. Unlike the expressive power seen on UML [11] and OWL 2 [20] with their breadth of features, the CML language and its extensible compiler intentionally support a limited number of features and scenarios.

This first version has been designed for the initial validation of the model-driven development approach taken by CML. As developers provide feedback, new language features may be iteratively added in order to enable the extensible CML compiler to support new modeling/development scenarios.

2.3 Extensible Target Generation

Some of the language features enable the generation of code into a wide range of target languages and technologies. Among the features that must be provided by the CML language, there is the ability to:

- break models into modules that can be shared as libraries;
- specify different code generation targets;
- and annotate model elements in order to provide more information for specific targets during code generation.

These features need to be available in a single language, they have to compatible with each other and with the code generation backend.

3 The Language

This section presents an overview of the conceptual modeling language. The *concrete syntax* will be presented using an example in subsection 3.1. In subsection 3.2, the example's model is shown as an *abstract syntax tree* parsed by the CML compiler. The mapping of the CML example to other modeling notations is illustrated in subsection 3.3. The CML metamodel (the *abstract syntax*'s structure) will be presented in subsection 3.4. (The appendix A provides a formal description of the *concrete syntax*, along with its mapping to the *abstract syntax*.)

3.1 An Example

On the example of figure 1, some concepts, such as *Book* and *Customer*, are declared in CML. The block-based syntax declaring each concept resembles the C [5] language's syntax. Each concept declares a list of properties. The property declarations are based on the Pascal [7] style for variable declarations, where the name is followed by a colon (":") and then the type declaration. Part of the CML syntax for expressions, such as the expression in *BookStore*'s *orderedBooks*, is based on OCL [10] expressions; while the syntax of the expression in *goldCustomers* is new, its semantics also match OCL [10] query expressions.

Some examples of the key language features shown in the figure 1 are:

- concepts: *Book* and *Customer*;
- attributes: *title* and *price* under the *Book* concept;
- derived attributes: *totalSales* under the *Customer* concept;
- uni-directional associations: *books* and *customers* under the *BookStore* concept;
- bi-directional associations: *CustomerOrder*, which binds two uni-directional associations (*orders* under the *Customer* concept and *customer* under the *Order* concept) into a single bi-directional association;
- derived associations: *goldCustomers* and *orderedBooks* under the *BookStore* concept.

These language features will be defined in the subsection 3.4.

```
concept BookStore
{
    books: Book+;
    customers: Customer*;
    orders: Order*;
    /goldCustomers = customers | select totalSales > 1000;
    /orderedBooks = orders.items.book;
}

concept Book
{
    title: String;
    price: Decimal;
    quantity: Integer = 0;
}

concept Customer
{
    orders: Order*;
    /totalSales = orders | collect result += total;
}

concept Order
{
    customer: Customer;
    total: Decimal;
}

association CustomerOrder
{
    Order.customer: Customer;
    Customer.orders: Order*;
}
```

Fig. 1. The example above is adapted to CML from the fictional Livir bookstore, which is presented as a case study in Wazlawick [21].

3.2 The Abstract Syntax Tree

The *BookStore* concept [4] specified in the model of figure 1, when parsed and instantiated by the CML compiler, generates the abstract syntax tree (or AST for short) in figure 2.

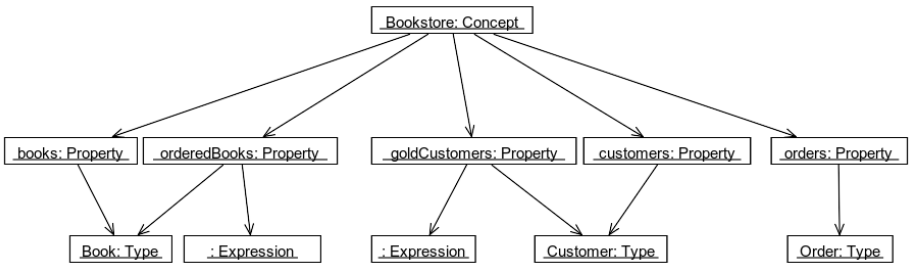

Fig. 2. The abstract syntax tree generated by the CML compiler after parsing the *BookStore* concept. TODO: Add "set" nodes to the AST.

The resultant AST in figure 2 is the CML compiler's internal representation of the *BookStore* concept, as specified by the model in the CML source listed in figure 1. Using EMOF's terminology [12], each node in figure 2 represents an instance of a class from the CML metamodel. The root-level node is an instance of the *Concept* class. Its subnodes are instances of the *Property* class, which in turn have (as subnodes) instances of the *Type* and *Expression* classes.

Once generated internally by the CML compiler, the AST in figure 2 will be used as the input for the extensible templates during the code generation phase, shown in section 4.

In order for the CML compiler to successfully generate the AST, it is necessary to define the CML metamodel, which is presented in subsection 3.4. Before, subsection 3.3 explores the mapping of conceptual models defined with CML to other modeling notations.

3.3 Mapping CML Source to UML and OCL

Part of the CML metamodel (presented in section 3.4) may be considered a small subset of the UML [11] metamodel. Thus, the structural (static) elements of CML models can be transformed into UML class diagrams. The example CML model in the listing of figure 1 is mapped to the UML model in figure 3.

In the UML class diagram of figure 3:

[4] For brevity, only displaying the partial tree of the CML model defined in figure 1 (just for the *BookStore* concept), but a similar tree is generated by the compiler for all other concepts.

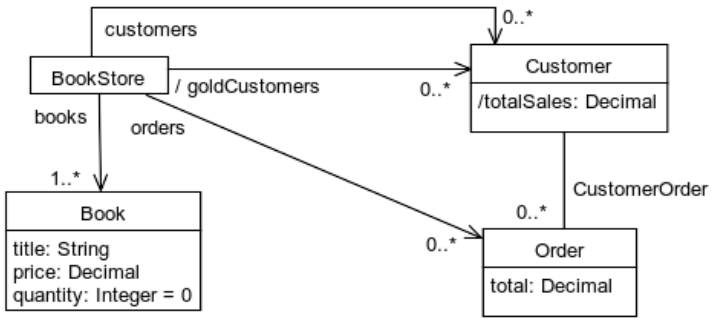

Fig. 3. The class diagram representing in UML [11] the same CML model listed in figure 1.

- the CML concepts (*BookStore*, *Book*, *Customer* and *Order*) are mapped to corresponding UML classes;
- the CML properties that represent attributes (such as *title*, *quantity* and *price* of *Book*) are mapped to corresponding UML attributes under each class;
- the CML properties that represent unidirectional associations (*books*, *customers*, and *goldCustomers* of *BookStore*) are mapped to UML associations with corresponding roles (showing the navigability direction, and matching the property names);
- the CML bidirectional association *CustomerOrder* (comprised by two CML properties: *Customer.orders* and *Order.customer*) is mapped to a UML association with bidirectional navigability (no direction arrow).

As demonstrated by this example, CML strives to enable modeling at the same conceptual level as allowed by UML. When compared to the UML metamodel, the CML metamodel supports only a core set of its elements (as shown in subsection 3.4). This minimally viable core has been intentionally designed to validate CML's objectives.

Besides the structural elements of a conceptual model (as seen above), CML also permits the use of expressions to set initial values to attributes, and to define derived properties for both attributes and associations. These expressions are partially based on the OCL [10] syntax, but they follow closely the OCL semantics.

For example, the following CML expression (extracted from figure 1) is a path-based navigation expression borrowed from OCL:

```
/orderedBooks = orders.items.book;
```

Using association properties, the expression above navigates from one instance of *BookStore*, passing through all linked *orders*, and then through all *items* of all *orders*, in order to return all books that have been ordered.

As another example, the following CML expression (also extracted from figure 1) does not follow the OCL syntax:

```
/goldCustomers = customers | select totalSales > 1000;
```

However, the expression above closely matches the semantics of the following OCL expression:

```
derive: customers->select(totalSales > 1000)
```

Both the CML expression and the OCL excerpt above evaluate to a set of *Customer* instances that have bought more than 1000 in the *BookStore*.

The OCL syntax for expressions that process collections of instances has the following general form:

```
collection->method_name(predicate or function)
```

The expression above is based on method invocations (an influence from UML's object-oriented paradigm), and thus it has an imperative style. CML has a different syntax (as shown previously), because it intends to be agnostic towards programming paradigms. Since CML uses collection expressions to define derived attributes and associations, its syntax to manipulate collections is more declarative, similar to SQL [6] or C#'s LINQ [17].

In CML, smaller expressions can also be combined into larger ones. For example:

```
/goldOrders = for order in bookStore.orders,
                            goldCustomer in bookStore.goldCustomers
              | select order.customer = goldCustomer
              | yield unique order
```

In the example above, all *orders* from *goldCustomers* are returned. In order to find them, the sub-expressions are sequentially evaluated as follows:

- the *for* expression provides a collection of all (*order*, *goldCustomer*) pairs;
- the *select* expression then keeps only the pairs that have matching customers;
- finally, the *yield* expression will map the collection of selected pairs into a set of unique *orders*.

Sub-expressions like *for*, *select* and *yield* can be combined in different configurations in order to derive any required attributes and associations.

3.4 The CML Metamodel (Abstract Syntax)

In the article *UML and OCL in Conceptual Modeling*, Gogolla [4] shows, by mapping the UML [11] metamodel to the ER [1] metamodel, how UML models (augmented by OCL [10] constraints) can be used to specify conceptual models.

Also, Wazlawick [21] systematically prescribes in his book a method for conceptual modeling using UML and OCL.

Since one key CML goal is enabling the specification of conceptual models (such as those specified by ER models and UML/OCL models), in order to present the key elements of the CML metamodel, a similar approach to Gogolla's is used to map the CML metamodel to the ER metamodel, and to the UML/OCL metamodel.

The EMOF [12] model presented by figure 4 is a simplified version of the CML metamodel:

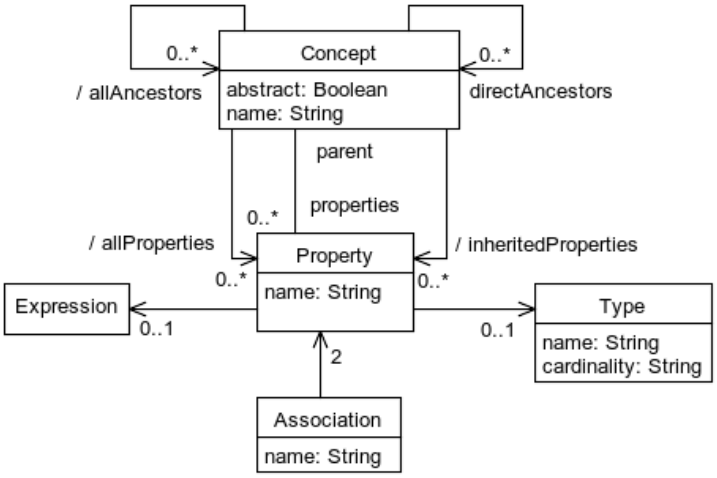

Fig. 4. This class diagram renders the EMOF [12] model defining the CML metamodel.

In CML metamodel shown in figure 4, a *Concept* is composed of zero-or-more *Property* instances. Each *Property* may have a *Type* and an *Expression*. If two *Property* instances represent the same bidirectional association, there must be an *Association* instance that binds them. Unidirectional associations are only represented by the *Property* instance (representing the association role) that enables the navigation from the source *Concept* instance to the target one (which is represented by the property's *Type*.)

Next, there is a description for the key metamodel elements.

Concept. According to Wazlawick [21], a concept represents complex information that has a coherent meaning in the domain. They aggregate attributes and cannot be described as primitive values. They may also be associated with other concepts. On the ER metamodel, it is known as *entity*; on the UML metamodel, as *class*.

Property. May be an attribute holding values of primitive types, or references (or collections of references) linking concepts. On the ER metamodel, the set of all references is known as *relationship*; on the UML metamodel, unidirectional associations. Attributes have the same name on all metamodels.

Association. Unlike the ER and UML metamodels, in the CML metamodel, only bidirectional associations are represented with the Association class. They bind the non-primitive properties (from the same or from different concepts.)

Type. They may be primitive types (such as Boolean, String, Decimal, and Regex), references to concept instances, or still collections of concepts. They may also be optional, meaning their value may or may not have been set.

4 The Extensible Compiler

In order to realize the CMP [2] manifesto's vision, the CML compiler generates code in any target language based on extensible templates. A set of core templates is provided by the CML compiler's base libraries. Third-party libraries can also provide their own templates, along with their conceptual models, in order to target specific technologies or platforms. Developers can also extend existing templates in order to adapt the implementation to characteristics specific to their projects.

Subsection 4.1 will provide an overview of the CML compiler's architecture. Next, subsection 4.2 will introduce the CML compiler's extensible templates. Finally, subsection 4.3 will lay out the CML compiler's mechanism for organizing and sharing conceptual models and extensible templates.

4.1 Compiler Overview

The CML compiler's overall architecture follows the standard compiler design literature [8]. An overview diagram of the architecture is shown in figure 5.

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

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 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, as shown in subsection 3.2. The AST serves then as the input for the *backend* component.

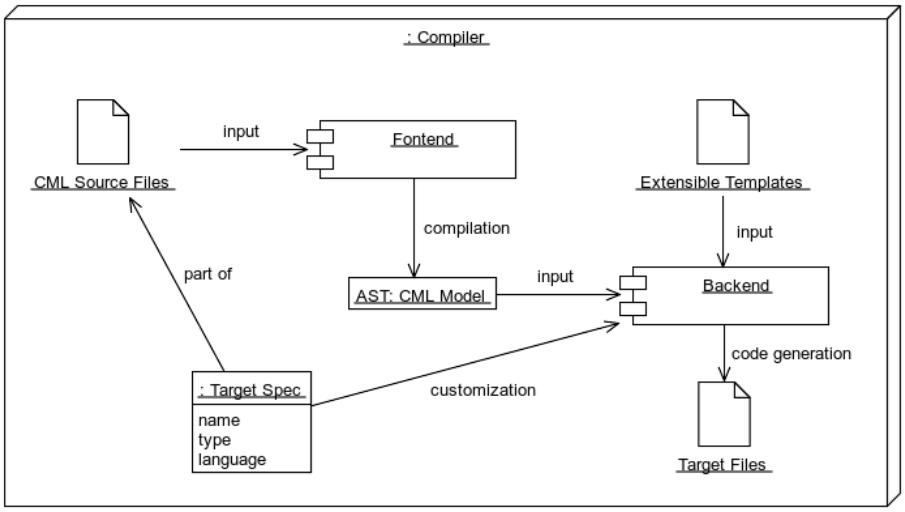

Fig. 5. An architectural overview of the CML compiler.

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, and it is presented in subsection 4.2.

4.2 Extensible Templates

Terence Parr has formalized and developed the StringTemplate [13] language for code generation. CML extensible templates are implemented in StringTemplate. 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* (introduced in subsection 4.1) in order to determine the file/directory names. Figure 6 shows an example of a *files template*.
- *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. Figure 7 shows a typical *content template*.

```
moduleFiles(target, module) ::= <<
maven:pom|pom.xml
>>

conceptFiles(target, concept) ::= <<
dto|src/main/java/<target.packagePath>/<concept.name>.java
>>
```

Fig. 6. An example of the *files template*.

```
import "/lang/class.stg"

dto(target, concept) ::= <<
package <target.packageName>;

import java.util.*;

public <class(concept)>
>>
```

Fig. 7. An example of the *content template*.

On the example of figure 6, two types of files are created for this target: one file for the CML module (named "pom.xml", and based on the "pom" template); and one for each concept found in the CML model (with the file extension ".java", and based on the "dto" template.)

On the example of figure 7, the "dto" content template is displayed, which can generate a DTO class in Java. The actual template that knows how to generate the class is imported from "/lang/class.stg".

4.3 Modules and Libraries

When developing a single application with just a few targets, having a single directory to maintain all the CML source code is sufficient. But, once more than one application is developed as part of a larger project, and CML model elements are shared among them, it is necessary to separate the common source code. Also, some applications cover different domains, and it may be beneficial to separate the source code into different CML models.

In order to allow that, CML supports *modules*. Grouping a set of CML model elements, a module in CML is conceptually similar to a UML [11] package. Physically, each module is a directory containing three sub-directories:

- *source*: where the CML source files reside.
- *templates*: optional directory containing templates for code generation.
- *targets*: created by the CML compiler to contain each target sub-directory, which in turn contains the target files generated for a given target.

Under the *source* directory, the module is defined by a *module specification*. If a module needs to reference CML model elements in other modules, then an import statement defines the name of the other modules. The CML compiler will then compile the imported modules before compiling the current module.

In order for the CML compiler to find the other modules, they must be in a sub-directory with the module's name in the same directory where the current module is placed.

CML modules have no versions as they are maintained in the same code repository with the other modules they import. However, one can package a module as a library, which will have a version and the same name as the module. This library in turn can be published into a public (or company-wide) *library site* in order to be shared with other developers.

A CML library is just a packaged, read-only module with a version of the format:

```
revision[.accretion][.fix]
```

Where semantic versioning [14] is followed:

- *revision* is the number of a library release incompatible with any previous releases with a lower revision number.
- *accretion* is the number of a library release compatible with any previous accretion number of the same revision.
- *fix* is the number of a library release that fixes an issue in a previous accretion.

Compatible versions do not change or remove public elements of the library's CML model (or function/parameters from the library's templates); only add new elements. Fixes cannot change the library's public elements; only internal elements. These rules may be enforced by the CML compiler when packaging and publishing new versions of a library.

5 Conclusion

The CML language and compiler make it possible to specify, in a single high-level language, the concepts of ever-changing, increasingly distributed software systems.

As opposed to implementing these concepts, their properties and associations in each target language, from a single CML model, the CML extensible templates generate code that keeps the implementations (across the different platforms and technologies) consistent with the specification. Also, as the technology landscape evolves, these textual CML models can be reused to generate code in new target languages and technologies.

The initial version of CML has been designed to validate this textual, model-driven approach of development. Practical application of CML in software development is needed in order to provide qualitative evidence that CML can indeed be used as a single source to implement multiple targets. Quantitative cost-benefit analysis (based on the implementation effort of hand-written vs generated lines-of-code, perhaps using a method adapted from the work of Gaffney et al [3]) may also provide data that shows whether the investment – made on the development of CML models – pays off. The data collected, together with the feedback provided by software developers, should then inform the iterative design of new CML features.

A The Language Specification

This appendix provides a formal description of the concrete syntax, along with its mapping to the abstract syntax.

A.1 Notation

The notation is based on BNF, but it also specifies the construction of the AST. Each grammar production that must be instantiated into a AST node is followed by a block specifying the node properties. The property may be assigned by a constant, by a terminal token, or by a reference to other nodes. Super-nodes may reference the sub-nodes (attribute synthesis in attributed grammars.) The sub-nodes may also reference super-nodes (attribute inheritance in attributed grammars.)

A.2 Extended Grammar

TODO: Update the grammar before submitting the article.

```
root Module: (Concept | Target)*
{
    elements = Concept* + Target*;
}

node Concept: 'abstract'? 'concept' NAME (':' AncestorList)? (';' | PropertyList)
{
    name = NAME;
    abstract = 'abstract'?;

    properties = Property*;
    properties.typeRequired = false;
    properties.typeAllowed = true;

    ancestors = for name in AncestorList.NAME* | map Model.concept[name];
    missingAncestors = AncestorList.NAME* - ancestors.name;
}

node Target: 'target' NAME PropertyList
```

```
{
    name = NAME;

    properties = Property*;
    properties.typeRequired = false;
    properties.typeAllowed = false;
}

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

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

node ancestorListNode: NAME (',' NAME)*;

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

token NAME: ('A'..'Z' | 'a'..'z') ( 'A'..'Z' | 'a'..'z' | '0'..'9' | '_' )*;
token STRING: '"' .*? '"';
token CARDINALITY: ('?' | '*');

function unwrap(str: String): String
{
    let /"?(<value>.*?)"?/ = str;

    return value or else str;
}
```

References

1. Chen, P.P.S.: The Entity-Relationship Model (Reprinted Historic Data). In: D.W. Embley, B. Thalheim (eds.) Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges, pp. 57–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). doi: 10.1007/978-3-642-15865-0_3. http://dx.doi.org/10.1007/978-3-642-15865-0_3
2. Embley, D.W., Liddle, S.W., Pastor, O.: Conceptual-Model Programming: A Manifesto. In: D.W. Embley, B. Thalheim (eds.) Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges, pp. 3–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). doi: 10.1007/978-3-642-15865-0_1. http://dx.doi.org/10.1007/978-3-642-15865-0_1
3. Gaffney Jr., J.E., Cruickshank, R.D.: A General Economics Model of Software Reuse. In: Proceedings of the 14th International Conference on Software Engineering, ICSE '92, pp. 327–337. ACM, New York, NY, USA (1992). doi: 10.1145/143062.143150. http://doi.acm.org/10.1145/143062.143150
4. Gogolla, M., Thalheim, B.: UML and OCL in Conceptual Modeling. In: D.W. Embley, B. Thalheim (eds.) Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges, pp. 85–122. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). doi: 10.1007/978-3-642-15865-0_4. http://dx.doi.org/10.1007/978-3-642-15865-0_4
5. ISO: ISO/IEC 9899:2011 Information technology — Programming languages — C. International Organization for Standardization (2011). https://www.iso.org/standard/57853.html
6. ISO: IEC 9075-1: 2003 (E) Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework) (2016). https://www.iso.org/standard/63555.html
7. Jensen, K., Wirth, N.: PASCAL User Manual and Report. Springer-Verlag New York, Inc. (1974)

8. Mogensen, T..: Introduction to Compiler Design. Undergraduate Topics in Computer Science. Springer (2011). http://dx.doi.org/10.1007/978-0-85729-829-4
9. OMG: Model Driven Architecture (MDA) MDA Guide rev. 2.0 (2014). http://www.omg.org/cgi-bin/doc?ormsc/14-06-01
10. OMG: Object Constraint Language (OCL), Version 2.4 (2014). http://www.omg.org/spec/OCL/2.4
11. OMG: Unified Modeling Language (UML), Superstructure, Version 2.5 (2015). http://www.omg.org/spec/UML/2.5
12. OMG: Meta Object Facility (MOF) Core Specification, Version 2.5.1 (2016). http://www.omg.org/spec/MOF/2.5.1
13. Parr, T.J.: Enforcing Strict Model-view Separation in Template Engines. In: Proceedings of the 13th International Conference on World Wide Web, WWW '04, pp. 224–233. ACM, New York, NY, USA (2004). doi: 10.1145/988672.988703. http://doi.acm.org/10.1145/988672.988703
14. Preston, T.: Semantic Versioning 2.0.0. http://semver.org
15. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Constructing Models with the Human-Usable Textual Notation. In: K. Czarnecki, I. Ober, J.M. Bruel, A. Uhl, M. Völter (eds.) Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, pp. 249–263. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). doi: 10.1007/978-3-540-87875-9_18. http://dx.doi.org/10.1007/978-3-540-87875-9_18
16. Thalheim, B.: The Theory of Conceptual Models, the Theory of Conceptual Modelling and Foundations of Conceptual Modelling. In: D.W. Embley, B. Thalheim (eds.) Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges, pp. 543–577. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). doi: 10.1007/978-3-642-15865-0_17. http://dx.doi.org/10.1007/978-3-642-15865-0_17
17. Torgersen, M.: Querying in C#: How Language Integrated Query (LINQ) Works. In: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07, pp. 852–853. ACM, New York, NY, USA (2007). doi: 10.1145/1297846.1297922. http://doi.acm.org/10.1145/1297846.1297922
18. Voelter, M.: Generic Tools, Specific Languages. Ph.D. thesis, Delft University of Technology (2014). http://repository.tudelft.nl/view/ir/uuid%3A53c8e1e0-7a4c-43ed-9426-934c0a5a6522
19. W3C: OWL 2 Web Ontology Language. Manchester Syntax (Second Edition) (2012). http://www.w3.org/TR/owl2-manchester-syntax
20. W3C: OWL 2 Web Ontology Language. Structural Specification and Functional-Style Syntax (Second Edition) (2012). http://www.w3.org/TR/owl2-syntax
21. Wazlawick, R.S.: Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL, and IFML. Morgan Kaufmann (2014). http://www.sciencedirect.com/science/book/9780124186736