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 presents a textual programming language for conceptual modeling (based on UML classes/associations and OCL constraints) and its compiler that can generate code in any target language or technology via extensible textual templates, both currently under initial stage of development. The language and compiler should allow the specification of information managed by ever-changing, increasingly distributed software systems. From a single source, automated code generation should keep implementations consistent with the specification across the different platforms and technologies. Furthermore, as the technology landscape evolves, the target templates may be extended to embrace new technologies. Unlike other approaches, such as MDA and MPS, the built-in tooling support, along with the textual nature of this modeling language and its extensible templates, is expected to facilitate the integration of model-driven software development into the workflow of software developers.

Keywords: conceptual modeling, 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, Model-Driven Architecture (MDA [21]) from 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 [23], OCL [22] and MOF [24]) to derive software artifacts and implementations via automated transformations. As one of its value propositions, the MDA guide [21] 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."

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 [36] in his *Generic Tools, Specific Languages* approach for model-driven software development. Voelter [36] has used domain-specific languages (DSLs) with the Metaprogramming System (MPS) in order to generate software artifacts. Unlike MDA, which is based on UML/MOF models, MPS allows the specification of models using domain-specific editors.

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. CML 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 [23] and OCL [22]; and, as *output*, any target languages, based on extensible templates that may be provided by the compiler's base libraries, by third-party libraries, or even by developers. As part of the author's Computer Sciences Bachelor Technical Report, both the CML language and compiler are in its initial stage of development, and available as an open source project online [28].

Section 2 explains the motivation for creating yet another language for conceptual modeling. The next two sections present the language (section 3) and the compiler with its extensible templates (section 4). Section 5 compares CML to other languages, tools and frameworks that can also generate code from conceptual models. We conclude in section 6, reiterating the objectives being pursued by CML and exploring options to validate the use of the CML compiler.

2 Why A New Language?

Thalheim [30] 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 text-based conceptual models into executable code of an extensible range of technologies. In order to achieve this purpose, a new language is designed with the following goals (among others):

- *Developer Experience*: CML follows the principle "the model is the code" as laid out on the *Conceptual-Model Programming* (CMP) manifesto [9]. Furthermore, CML is also intended to enable software developers to do *conceptual modeling* on the same workflow they are used to doing *programming*; that is, using text editors and a compiler. CML strives to *not only be* the

[3]Projectional editors in MPS do not rely on parsers. Instead, the abstract syntax tree (AST) is modified directly. MPS renders the visual representation of the AST based on the DSL editor definition.

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 [23] notation, on the other hand, being graphical, is not suited for mainstream, textual programming. However, the *Human Usable Textual Notation* (HUTN) [26] is a textual syntax for MOF-based [24] 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.
- *Language Evolution*: This initial version is being designed for the validation of the model-driven development approach offered by CML. Unlike the expressive power seen on UML [23] and OWL 2 [37] with their breadth of features, the CML language initially supports generalization/specialization, bidirectional associations (with zero-or-one and zero-or-many cardinality) and the ability to define derived attributes and associations with OCL-like expressions. These features have already allowed the specification of CML compiler's own metamodel in CML itself. The CML compiler is thus the first system used to validate CML's aplicability, and will continue to do so as the language evolves.
- *Extensible Target Generation*: Some of the language features should 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, it is the ability to break models into modules (already available); the ability to share modules as libraries (planned); the ability to specify different code generation targets (already available); and the ability to annotate model elements in order to provide more information for specific targets during code generation (also planned). In order to effectively support code generation, these language features must be available in a single language, so that they can be compatible with each other and with the compiler backend.

Section 5 provides further motivation for developing CML, comparing it to related work.

3 The Language

This section presents an overview of the conceptual modeling language. The *concrete syntax* is presented using an example in subsection 3.1. The mapping of the CML example to UML [23] and OCL [22] is illustrated in subsection 3.2. The CML metamodel (the *abstract syntax*'s structure) is presented in subsection 3.3. (The *CML Specification* [28] 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

[15] language's syntax. Each concept declares a list of properties. The property declarations are based on the Pascal [17] 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 [22] expressions. While the syntax of the expression in *goldCustomers* is new, its semantics also match OCL [22] query expressions.

```
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 = sum(orders.total);
}

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

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

Fig. 1. Adapted from the fictional Livir bookstore; a case study by Wazlawick [38].

The key language features are: *Book* and *Customer* are concepts; *title* and *price* under the *Book* concept are attributes; *totalSales* under the *Customer* concept is a derived attribute; the properties *books* and *customers* declared under the *BookStore* concept represent unidirectional associations (in UML [23], they would correspond to the association roles); *CustomerOrder* binds two unidirectional associations (represented by the *orders* property under the *Customer* concept and by the *customer* property under the *Order* concept) into a single bidirectional association; the properties *goldCustomers* and *orderedBooks* under the *BookStore* concept are examples of derived associations.

These language features are defined in the subsection 3.3.

3.2 Mapping CML Source to UML and OCL

Part of the CML metamodel (presented in section 3.3) may be considered a small subset of the UML [23] 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 2.

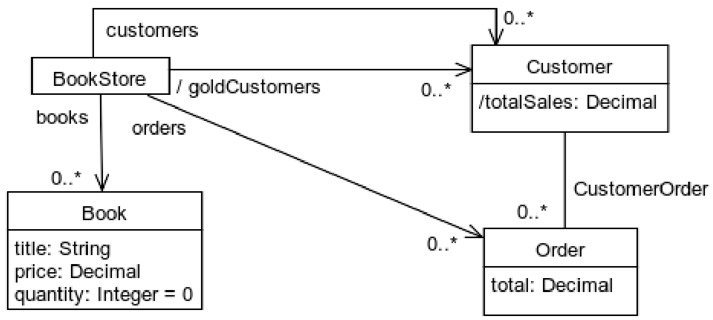

Fig. 2. The UML class diagram [23] for the CML model listed in figure 1.

In figure 2, 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 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 and cardinality.) The CML bidirectional association *CustomerOrder* (comprised by two CML properties: *Customer.orders* and *Order.customer*) is mapped to a UML association with bidirectional navigability (that is, no direction arrow.) As demonstrated by this example, CML strives to enable modeling at the same conceptual level as allowed by UML. That being said, when compared to the UML metamodel, the CML metamodel supports only a core set of its elements, as shown in subsection 3.3.

Besides the structural elements of a conceptual model (as seen above), CML also has expressions that can set initial values to attributes, and define derived properties for both attributes and associations. CML expressions are partially based on the OCL [22] 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, on the other hand, intends to be agnostic towards programming paradigms. By using extensible comprehensions [33] to define derived attributes and associations, CML's syntax is more declarative, similar to SQL [16] or C#'s LINQ [32]. 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: order
```

Above, all *orders* from *goldCustomers* are returned. The sub-expressions are evaluated sequentially: the *for* expression provides a cross join of all (*order*, *goldCustomer*) pairs; the *select* expression selects only the pairs that have matching customers; Finally, the *yield* expression maps selected pairs into a sequence of *orders*. Sub-expressions like *for*, *select* and *yield* can be combined in different configurations in order to derive any required attributes and associations.

3.3 The CML Metamodel (Abstract Syntax)

In the article *UML and OCL in Conceptual Modeling*, Gogolla [12] shows, by mapping the UML [23] metamodel to the ER [8] metamodel, how UML models (augmented by OCL [22] constraints) can be used to specify conceptual models. Also, Wazlawick [38] systematically prescribes 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 [24] model presented by figure 3 is a simplified version of the CML metamodel. As shown, a *Concept* is composed of zero-or-more *Property* instances. Each *Property* must have a *Type* and an optional *Expression*. If two *Property* instances represent both ends of the same bidirectional association, there must be an *Association* instance that binds them. Unidirectional associations are only represented by a single *Property* instance (actually representing

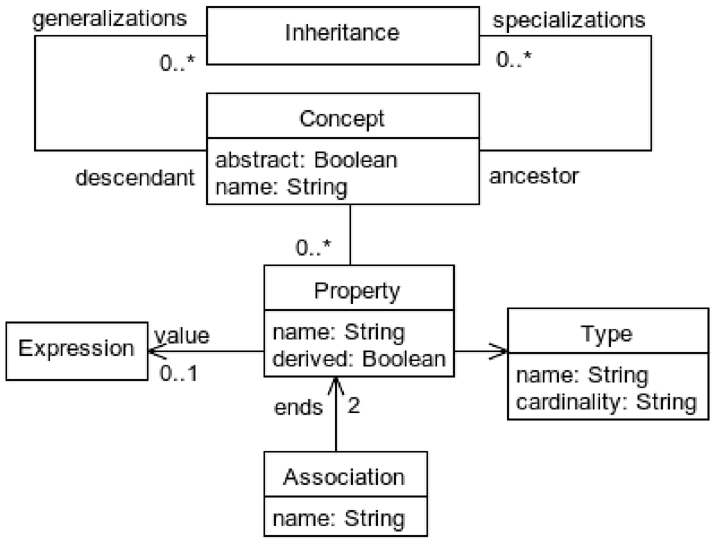

Fig. 3. Simplified EMOF [24] model defining the CML metamodel.

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 [38], 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 Type*; on the UML metamodel, as *Class*. CML's *Concept* differs, however, from the UML *Class*, because it has only *Property* instances, while the UML *Class* may also have *Operation* instances.
- *Property*: May hold values of primitive types, in which case they represent an attribute on the *ER* and *UML* metamodels; or may hold references (or collections of references) linking to instances of other concepts. On the ER metamodel, a set of all references linking one *Entity Type* to another is known as a *Relationship*; on the UML metamodel, it is known as a unidirectional *Association*.
- *Association*: Unlike the ER and UML metamodels, in the CML metamodel, only a bidirectional *Association* is represented with the *Association* class. Using UML terminology, they bind the reference (non-primitive) properties (of the same, or of different concepts), so that the *Association* links are accessible from each participating association end. It directly represents in the CML metamodel what normally requires additional implementation in

programming languages. It is inspired[4] on the work of Cardoso [7], which extends the C# language to represent bidirectional associations; it is also inspired on the work of Balzer et. al. [2], which uses *member interposition* to model relationships.

- *Type*: They may be of: a primitive type (such as Boolean, String, and Decimal); a reference to a *Concept* instance (*cardinality* equal to one); a sequence of references (*cardinality* equal to zero-or-many); or optional, meaning their value may or may not have been set (also defined by the *cardinality* property). CML also supports the tuple and lambda types, which are used in expressions.
- *Inheritance*: Following the the UML [23] metamodel, CML provides the generalization/specialization relationship. In CML, a *Concept* may be a specialization of two or more other *Concept* instances. If a *Property* has been defined by more than one generalization, CML requires it to be redefined by the specialization in order to resolve the definition conflict.

4 The Extensible Compiler

In order to realize the CMP [9] manifesto's vision, the CML compiler can generate code in any target language if the corresponding templates are provided. A set of core templates is provided by CML compiler's base module, which is currently supporting Java and Python. In order to target specific technologies or platforms, third-party modules can also provide their own templates, along with their conceptual models. Developers can also extend existing templates in order to adapt the implementation to characteristics specific to their projects.

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

4.1 Compiler Overview

An overview diagram of the architecture is shown in figure 4. The two main components of the compiler, and the artifacts they work with, are presented below:

- *Frontend:* receives as input the *CML source files*. It parses the files into an internal representation of the *CML model*. Syntactical and semantic validations are then executed. Any errors are presented to the developer, interrupting the progress to the next phase. If the *source files* are parsed and validated successfully, the *CML model* serves then as the input for the *backend* component.

[4] The syntax used in CML resembles the syntax of a *struct* in C [15], while Cardoso [7] uses a verbose syntax. Also, unlike CML, Cardoso does *not* bind properties that represent each association end; instead, associations – unidirectional or bidirectional – are declared independently of class properties.

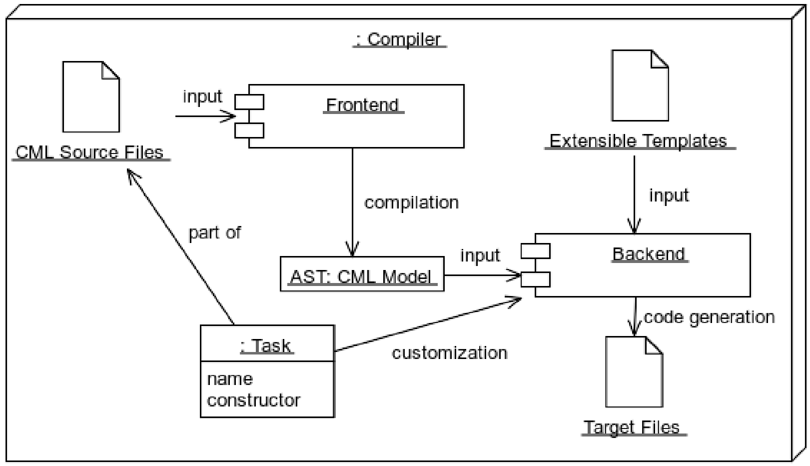

Fig. 4. An architectural overview of the CML compiler.

– *Backend:* receives the *CML model* as input. Based on the *constructor* defined by a *task*, the *backend* chooses which *extensible templates* to use for code generation. The *target files* are then generated to be consumed by other tools. The *task* and associated *constructor* play a key role in determining the kind of *target* to be generated.

4.2 Extensible Templates

Parr has formalized and developed the StringTemplate [25] language for code generation. CML's 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 *constructor* 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 *task* (introduced in subsection 4.1) in order to determine the file/directory names. An example of a *files template* is shown below:

```
model_files(task, model) ::= <<
pom_file|pom.xml
>>

concept_files(task, concept) ::= <<
concept_file|src/main/java/<task.packagePath>/<concept.name; format="pascal-case">.java
>>
```

– *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. A typical *content template* is shown below:

```
import "/design/poj.stg"

concept_file(task, concept) ::= <<
package <task.packageName>;

import java.util.*;

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

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

On the *content template* example, the "concept_file" content template is displayed, which can generate a *Data Type Object* (DTO) class in Java. The actual template that knows how to generate the class is imported from "/design/poj.stg".

4.3 Modules and Libraries

When developing a single application with just a single target language, 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 [23] package. Physically, each module is a directory containing the following sub-directories:

- *source*: where the CML source files reside.
- *templates*: optional directory containing templates for code generation.
- *tests*: optional directory containing tests that verify the generated code.
- *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.

A CML module have no version as it is maintained in the same code repository with the other modules it imports. However, it is planned that a future version of CML will allow packaging a module as a library, which will have a version and the same name as the module. Such a library will in turn be published into a public (or company-wide) *library site* in order to be shared with other developers. A CML library is expected to become a packaged, read-only, versioned module.

5 Related Work

This section compares CML to other languages, tools and frameworks that can also generate code from conceptual models. Each paragraph covers a different category, enumerating specific solutions and characterizing their relevance to CML, and also their differences.

When compared to CML, the text-based languages are the most relevant. MPS [36] is a development environment for DSLs. Strictly speaking, its DSLs are not textual, since their AST is directly edited on projectional editors. However, the editors allow textual representations. Unlike MPS, the DSLs created with the M language [6] are truly textual. It was part of the discontinued Oslo project from Microsoft, which incorporated into Visual Studio similar capabilities to what is available on MPS. Xtext/Xtend [3] allows the definition of textual DSLs to generate code from conceptual models edited on Eclipse. It is similar to the Oslo project from Microsoft, and based on EMF [29]. MM-DSL [35], on the other hand, allows the definition of metamodels (abstract syntax; not the actual DSLs), which serve as input to generate domain-specific modeling tools. ThingML [14] is also a language and code generation framework for the development of software in embedded devices. XML may also be used for conceptual modeling, and XSLT then used to create the templates for code generation, as shown by Gheraibia et all [11]. Observe that most of the solutions previously mentioned enable modeling via DSLs, while CML is a generic language for modeling in any domain.

Graphical languages also have some relevance to CML, despite the latter being a textual language, because the former have also been used to generate code in other target languages. MPS [36], besides the textual models, also allows the creation of graphical models. FCML [18], on other hand, incorporates and extends conceptual modeling languages (ER, UML, and BPMN) via the OMNILab tool in order to generate code. MetaEdit+ [31] is another development environment that allows the creation of modeling tools and code generators for visual DSLs. As mentioned previously in this article, MDA [21] is the initiative from OMG to use UML [23] for model-driven development. IFML [5] is an example from OMG of a high-level language that can be used to generate user interfaces on different platforms, such as the Web, or on mobile devices. The major drawback of graphical languages, as covered in section 2, is their difficulty to integrate seamlessly with the workflow, tools and mindset of software developers.

Frameworks also allow code generation from conceptual models, but lack a modeling language – graphical or textual. EMF [29] is a classical example, where modeling is done via editors on Eclipse or via a programming interface, and the models are stored in the ECORE/XML format. Frameworks may also be used as the infrastructure of modeling languages. EMF, for example, is the framework supporting Xtext [3]. Conceivably, other modeling languages may also target EMF. In fact, CML's extensible compiler allows the implementation of templates that target EMF.

As seen in previous sections, the CML compiler uses StringTemplate [25] as the language for its code generation templates. There are other template languages designed for code generation from conceptual models. Xpand [13] allows the definition of templates with multiple variability regions. EGL [27] is another language that allows code generation from models. MOFScript [20] allows code generation from models defined by any type of metamodel. JET [34] allows code generation from EMF [29] models. One strength of StringTemplate is its extensibility mechanisms. It is possible to define a core set of templates that define patterns, and then extend them with the specifics of each target language or technology. It is also possible to share templates as libraries, which can be further extended for specific purposes by third-parties. Xpand also allows this level of extensibility.

Just like CML, there are programming languages that provide the ability to declare bidirectional associations. DSM [2] is an object-oriented programming language with support for associations. Fibonacci [1] is programming language for object-oriented databases that allows the modeling of association roles. ASSOCIATION# [7], on the other hand, is an extension to C# that allows the modeling of associations. Likewise, RelJ [4] is a Java extension with support for associations. One key drawback of these languages is the fact that their conceptual models cannot be reused to generate code in any other language or technology; they are, for all intents and purposes, the target language.

There are also other conceptual languages whose original focus has not been to support code generation or implementation, but to serve solely as modeling artifacts. Languages, such as OWL [37] and Telos [19], have been designed as ontology metamodels to support the representation and storage of knowledge, and to allow automated reasoning from knowledgebases; OWL being the *lingua franca* of the semantic web, while Telos has been created to store ontologies in a object-oriented database. Other languages, like UML [23] and ER [8], have been originally intended as tools to support the analysis and design of software systems, and only later have been repurposed for model-driven software development. The relevance of these languages to CML comes from the expressivity power their metamodels provide for conceptual modeling. For that reason, CML should continue to expand its capabilities by borrowing features from these languages.

6 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 modeling 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 software development. Practical application of CML 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 [10]) 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.

References

1. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: VLDB, vol. 93, pp. 39–51 (1993)
2. Balzer, S., Gross, T.R., Eugster, P.: A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In: E. Ernst (ed.) ECOOP: 21st European Conference. Proceedings, pp. 323–346. Springer (2007)
3. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)
4. Bierman, G., Wren, A.: First-Class Relationships in an Object-Oriented Language. In: A.P. Black (ed.) ECOOP 2005: 19th European Conference. Proceedings, pp. 262–286. Springer (2005)
5. Brambilla, M., Mauri, A., Umuhoza, E.: Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End. In: I. Awan (ed.) MobiWIS 11th. Proceedings, pp. 176–191. Springer (2014)
6. Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards Model Driven Tool Interoperability. In: T. Kühne (ed.) ECMFA 6th. Proceedings, pp. 32–47. Springer (2010)
7. Cardoso, I.S.: Inserindo suporte a declaração de associações da UML 2 em uma linguagem de programação orientada a objetos. Master's thesis, Universidade Federal de Santa Catarina (2011)
8. Chen, P.P.S.: The Entity-Relationship Model (Reprinted Historic Data). In: D.W. Embley (ed.) Handbook of Conceptual Modeling, pp. 57–84. Springer (2011)
9. Embley, D.W., Liddle, S.W., Pastor, O.: Conceptual-Model Programming: A Manifesto. In: D.W. Embley (ed.) Handbook of Conceptual Modeling, pp. 3–16. Springer (2011)
10. 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)
11. Gheraibia, Y., Bourouis, A.: Ontology and automatic code generation on modeling and simulation. In: 6th SETIT. Proceedings, pp. 69–73 (2012)
12. Gogolla, M., Thalheim, B.: UML and OCL in Conceptual Modeling. In: D.W. Embley (ed.) Handbook of Conceptual Modeling, pp. 85–122. Springer (2011)

13. Greifenberg, T., Müller, K., Roth, A., Rumpe, B., Schulze, C., Wortmann, A.: Modeling Variability in Template-based Code Generators for Product Line Engineering. CoRR **abs/1606.02903** (2016)
14. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: Thingml: A language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th MODELS, pp. 125–135 (2016)
15. ISO: ISO/IEC 9899:2011 Programming languages — C. International Organization for Standardization (2011)
16. ISO: IEC 9075-1: 2003 (E) Database languages — SQL Part 1: Framework (SQL/Framework) (2016)
17. Jensen, K., Wirth, N.: PASCAL User Manual and Report. Springer-Verlag (1974)
18. Karagiannis, D., Buchmann, R.A., Burzynski, P., Reimer, U., Walch, M.: In: Domain-Specific Conceptual Modeling: Concepts, Methods and Tools, pp. 3–30. Springer (2016)
19. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing Knowledge About Information Systems. ACM TIS **8**(4), 325–362 (1990)
20. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.J.: Toward Standardised Model to Text Transformations. In: A. Hartman (ed.) ECMDA-FA. Proceedings, pp. 239–253. Springer (2005)
21. OMG: Model Driven Architecture (MDA) Guide rev. 2.0 (2014)
22. OMG: Object Constraint Language (OCL), Version 2.4 (2014)
23. OMG: Unified Modeling Language (UML), Superstructure, Version 2.5 (2015)
24. OMG: Meta Object Facility (MOF) Core Specification, Version 2.5.1 (2016)
25. 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)
26. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Constructing Models with the Human-Usable Textual Notation. In: K. Czarnecki (ed.) MoDELS: 11th International Conference. Proceedings, pp. 249–263. Springer (2008)
27. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: I. Schieferdecker (ed.) ECMDA-FA: 4th European Conference. Proceedings, pp. 1–16. Springer (2008)
28. dos Santos, Q.C.M.: CML Project (2017). http://github.com/orgs/cmlang
29. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
30. Thalheim, B.: The Theory of Conceptual Models, the Theory of Conceptual Modelling and Foundations of Conceptual Modelling. In: D.W. Embley (ed.) Handbook of Conceptual Modeling, pp. 543–577. Springer (2011)
31. Tolvanen, J.P.: MetaEdit+: Domain-specific Modeling for Full Code Generation Demonstrated. In: Companion to the 19th Annual ACM SIGPLAN Conference, OOPSLA '04, pp. 39–40. ACM, New York, NY, USA (2004)
32. Torgersen, M.: Querying in C#: How Language Integrated Query (LINQ) Works. In: Companion to the 22Nd ACM SIGPLAN Conference, OOPSLA '07, pp. 852–853. ACM, New York, NY, USA (2007)
33. Trinder, P.: Comprehensions, a query notation for dbpls. In: Proceedings of DBPL3, pp. 55–68. Morgan Kaufmann Pub. Inc., San Francisco, USA (1992)
34. van Emde Boas, Ghica: Template programming for model-driven code generation. In: 19th Annual ACM SIGPLAN Conference (2004)
35. Visic, N., Karagiannis, D.: Developing Conceptual Modeling Tools Using a DSL. In: R. Buchmann (ed.) KSEM: 7th International Conference. Proceedings, pp. 162–173. Springer (2014)

36. Voelter, M.: Generic Tools, Specific Languages. Ph.D. thesis, Delft University of Technology (2014)
37. W3C: OWL 2 Structural Specification and Functional-Style Syntax (Second Edition) (2012)
38. Wazlawick, R.S.: Object-Oriented Analysis and Design for Information Systems. Morgan Kaufmann (2014)