Recurring Return on Modeling Investment: A Conceptual Modeling 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. 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, which is expected to promote its adoption. A cost-benefit analysis model is also provided, which should assist the stakeholders in measuring the return of their investment in modeling.

Keywords: conceptual modeling, omg, mda, uml, ocl, mof, mps, mde, mdsd, er, entity-relationship model, programming language, compiler, code generation, model-driven software development, model-driven engineering, modeling investment, classes, associations, constraints, specification, software tools, 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 [3]) 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 [5], OCL [4]

and MOF [6]) to derive software artifacts and implementations via automated transformations. As one of its value propositions, the MDA guide [3] 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 [7] in his *Generic Tools, Specific Languages* approach for model-driven software development. Voelter [7] 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 [5] and OCL [4];
- 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 the developers themselves.

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). The section 5 discusses how CML can be integrated into the development workflow. The Section 6 provides a cost-benefit analysis model that can be used by organizations in order to determine whether integrating CML into its development workflow would provide recurring return on their modeling investment. We conclude in section 7, reiterating the objectives achieved by CML and exploring future possibilities to expand CML's reach.

[3] As explained by Voelter [7], 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 Why A New Language?

At this point, one could question: why define a new textual language for conceptual modeling from scratch? Instead of basing it on an standard, richer textual language, such as OWL 2 [9]. There are two primary reasons explained in the follow subsections:

2.1 Developer Experience

CML is designed for software developers. It is intended to enable developers to do *conceptual modeling* much the same way they are used to doing *programming*.

The Manchester [8] syntax of OWL 2 is intended to be user-friendly, but it still does not resemble the syntax of commonly used programming languages, such as C [1] and its syntax-alike descendants.

Using CML, and its familiar syntax (as we shall demonstrate in the next sections), it is expected that developers will be able to raise the abstraction level of their programs, and still work with a syntax that is familiar to them. By providing its own syntax, CML then promotes the *modeling-as-programming* approach being presented by this work.

2.2 Language Evolution

CML is intended to evolve with its compiler, and the tooling around it. Unlike the expressive power seen on OWL 2 [9] with its breadth of features, and its ambition to model complete ontologies for the Web, the CML language and its extensible compiler intentionally support a limited number of features and scenarios.

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

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

Before exploring the abstract syntax, a concrete example is displayed in figure 1 and commented below:

```
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. This example is adapted to CML from the fictional Livir bookstore, which is presented as a case study in Wazlawick [10].

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 [1] language's syntax. Each concept declares a list of properties. The property declarations are based on the Pascal [2] 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 [4] expressions; while the syntax of the expression in *goldCustomers* is new, even though its semantics also match OCL [4] 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.

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 following abstract syntax tree (or AST for short):

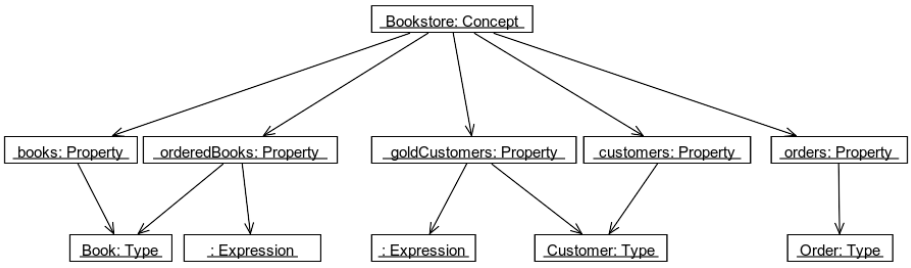

Fig. 2. The abstract syntax tree generated by the CML compiler after parsing the *BookStore* concept.

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 [6], 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.

[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.

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

3.3 Mapping CML to Other Modeling Notations

The model above can also be represented as the following ER model:

[ER model]

The model can also be shown as the following UML class diagram:

[UML Class Diagram]

3.4 The CML Metamodel (Abstract Syntax)

The focus of this language is to enable the specification of conceptual models, such as those specified by ER [reference] models and UML [reference] class diagrams. In [article], [Author] shows how UML models, augmented by OCL constrains, can be used to specify conceptual models by mapping their metamodel to the ER metamodel. In order to present the CML concepts in the following subsections, a similar approach will be used by mapping the CML metamodel to the UML/OCL metamodels.

Below, you can see an overview of the CML metamodel in a UML class diagram (footnote: now used to model the CML metamodel, as opposed to modeling an instance of a CML source file — a CML model), where each class represents a CML concept:

[diagram]

[Describe the relationships between the concepts in the diagram]

[Subsection for each key concept in the CML metamodel.]

4 The Extensible Compiler

5 The Development Workflow

6 The Cost-Benefit Analysis Model

7 Conclusion

A The Concrete/Abstract Syntax

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

References

1. ISO: ISO/IEC 9899:2011 Information technology — Programming languages — C. International Organization for Standardization (2011). URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853
2. Jensen, K., Wirth, N.: PASCAL User Manual and Report. Springer-Verlag New York, Inc. (1974)
3. OMG: Model Driven Architecture (MDA) MDA Guide rev. 2.0 (2014). URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01
4. OMG: Object Constraint Language (OCL), Version 2.4 (2014). URL: http://www.omg.org/spec/OCL/2.4
5. OMG: Unified Modeling Language (UML), Superstructure, Version 2.5 (2015). URL: http://www.omg.org/spec/UML/2.5
6. OMG: Meta Object Facility (MOF) Core Specification, Version 2.5.1 (2016). URL: http://www.omg.org/spec/MOF/2.5.1
7. Voelter, M.: Generic Tools, Specific Languages. Ph.D. thesis, Delft University of Technology (2014). URL: http://repository.tudelft.nl/view/ir/uuid%3A53c8e1e0-7a4c-43ed-9426-934c0a5a6522
8. W3C: OWL 2 Web Ontology Language. Manchester Syntax (Second Edition) (2012). URL: http://www.w3.org/TR/owl2-manchester-syntax
9. W3C: OWL 2 Web Ontology Language. Structural Specification and Functional-Style Syntax (Second Edition) (2012). URL: http://www.w3.org/TR/owl2-syntax
10. Wazlawick, R.S.: Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL, and IFML. Morgan Kaufmann (2014). URL: http://www.sciencedirect.com/science/book/9780124186736