The XModeler^{ML}

A language engineering, modeling and execution environment

Ulrich Frank[1,*], Tony Clark[2]

[1]University of Duisburg-Essen, Germany
[2]Aston University, Birmingham, UK

Abstract
This paper presents the versatile language engineering, modeling, and execution environment XModeler^{ML}. Starting with the motivation for developing the tool and a brief historical background, the underlying language engineering technology is presented. Subsequently, we give an overview of core concepts of the (meta) modeling language FMML^{x} that is used for creating models and DSMLs with the XModeler^{ML}. Finally, an outline of the components of the XModeler^{ML} gives an impression of how to use the tool.

Keywords
multi-level modeling, reflexive language architecture, DSML development, model-based development

1. Introduction

The development of languages, models, and corresponding tools and applications is a demanding undertaking. It often requires expertise and resources that are beyond many organizations capabilities. Reuse is a promising approach to reduce the corresponding effort without the need to compromise on quality. However, reuse is not easy to achieve, since languages, models and applications in general often have to satisfy specific requirements of particular use cases. Apart from reusing existing components, the only way to achieve reuse is to aim at powerful abstractions, which capture commonalities of a wide range of artefacts and, at the same time, support convenient and safe adaptation to specific needs. There are various approaches in software engineering and conceptual modeling that pursue this objective. The abstraction they allow for often depends on the (programming) language they make use of or they were designed for. This is especially the case for the fact that most (object-oriented) languages allow for one classification level only. As a consequence, more abstract knowledge about a class of systems or a domain that would require higher levels of classification can hardly be expressed.

The tool we present in this paper, the XModeler^{ML}, is based on a reflexive language architecture that overcomes this limitation. It provides powerful abstractions over languages, models and tools that do not only support the efficient realization of custom languages and tools, but also enable new system architectures that promote reuse, adaptability, integration and user empowerment. The development of the XModeler^{ML}started in 2010 within the project "Language Engineering for Multi-Level Modeling" (LE4MM) as a collaboration between the universities Duisburg-Essen and Middlesex, London [1]. The foundational language model and the corresponding tool, the XModeler, had been developed earlier [2], [3].

The design of the XModeler^{ML}was motivated by objectives, which to achieve was hardly possible with existing language technologies. The presentation starts with a discussion of these objectives. We will then give an overview of the foundational architecture and present essential benefits from using the XModeler^{ML}.

Companion Proceedings of the 16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling and the 13th Enterprise Design and Engineering Working Conference, November 28 – December 1, 2023, Vienna, Austria
*Corresponding author.
ulrich.frank@uni-due.de (U. Frank); tony.clark@aston.ac.uk (T. Clark)
0000-0002-8057-1836 (U. Frank); 0000-0003-3167-0739 (T. Clark)

2. Motivation

The main reason for choosing the language technology implemented in the XModeler was frustration with prevalent language architectures. Previously the enterprise modeling group at the University of Koblenz and later at the University of Duisburg-Essen had developed a range of languages for enterprise modeling. To foster the use of these languages for analysis and design purposes, various corresponding modeling tools had been developed.

At first, Smalltalk was chosen to this end [4]. With its clean and consequent object-oriented foundation and its pathbreaking development environment it proved to serve as a powerful instrument for the implementation of modeling tools. In addition to classes, Smalltalk provides metaclasses. In principal, these are useful since they allow for additional abstraction. However, the abstraction enabled by metaclasses in Smalltalk is limited: by default, every class in Smalltalk has one metaclass which in turn has exactly one instance only. As a consequence, it is not possible to define metaclasses of a range of classes – or even of metaclasses. This limitation is a serious obstacle to the development and use of modeling tools (see below). Later, EMF, Eclipse amd Java were used for a further generation of tools [5], mainly for the reason that they provide for platform independence and offer extensive libraries, which promote development productivity. Apart from that, Java represented a step backwards compared to Smalltalk.

The frustration accumulated through the experience with Java and even with Smalltalk was caused primarily by the following limitations. For an extensive analysis of limitations inherent to the traditional language paradigm see [6].

Lack of expressiveness: It is a pivotal guideline, both for the design of modeling languages and for conceptual modeling in general to express all knowledge one has about a domain at the highest level of abstraction in order to avoid redundancy. It happens frequently that this is not possible with (meta-) modeling languages based on the MOF. The example in Fig. 1 illustrates this limitation. We know that a master thesis is a kind of document. At first, it seems plausible to regard the class **`MasterThesis`** as being specialized from the class **`Document`**. However, the attribute **`maxPages`** in **`Document`** is not to be inherited to **`MasterThesis`**. Instead, it should be instantiated there to indicate the maximum numbers of pages defined for master theses. Also, we know of documents that particular instances have a page count. Therefore, one should represent this knowledge with the specification of the class **`Document`**. However, it is not possible to define that it is to be instantiated only at level 0.

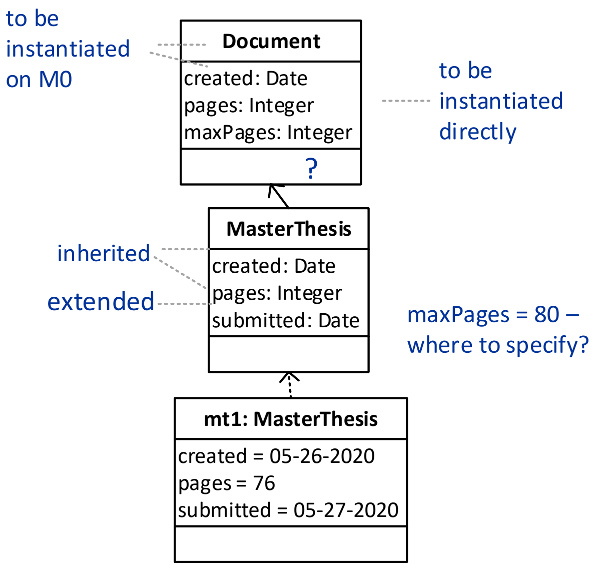
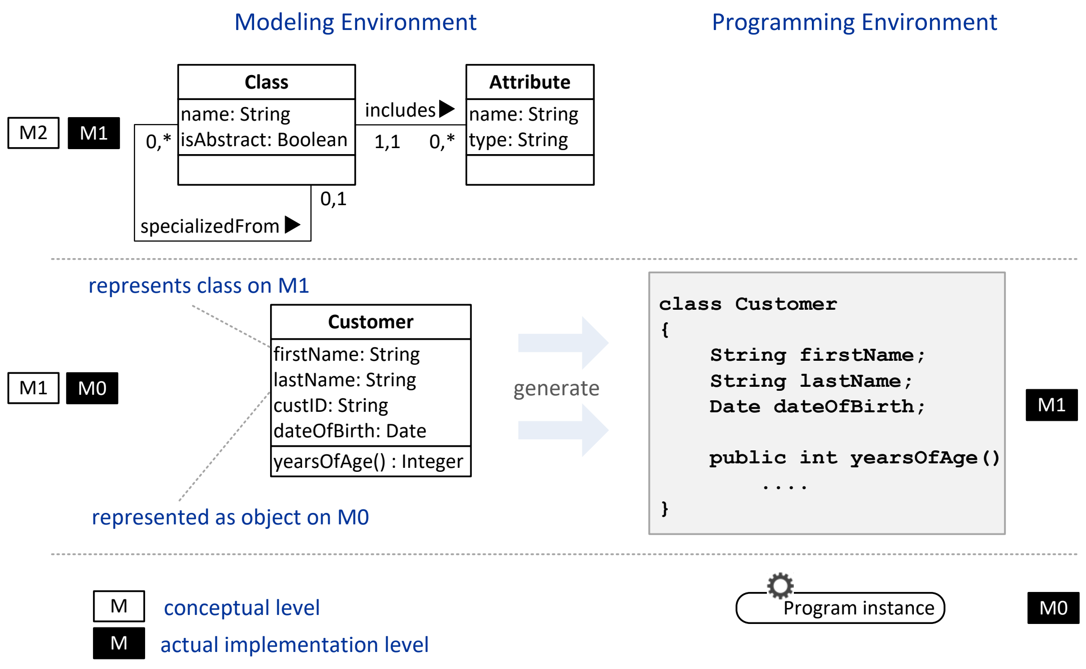


Figure 1: Illustration of limited expressiveness in traditional paradigm

Limitations of DSML design: This is a special case of the previous limitation. It occurs frequently with the design of modeling languages and is especially annoying, if the models created with a modeling language should be further instantiated. To this end, a process modeling language should include the knowledge we have about specific properties of process instances. For example, every process instance has a start time and a termination time. Apparently, this obvious knowledge cannot be expressed with a metaclass that is used for the specification of a modeling language.

Pitfalls of model-driven development: Model-driven software development is an appealing idea. It advocates to focus on modeling and do without coding as much as possible. Finally, when the models are complete, code is generated. While this seems like a convincing approach to software development, it suffers from a serious drawback: during its lifetime, a software system has to be adapted to new requirements. No matter whether changes are applied to the code – which will often be the case – or to the corresponding models, in any case both representations need to be synchronized, if one does not want to give up on the benefits of having an up to date model.

The example in Fig. 2 makes clear why code generation is necessary in the traditional language paradigm. While the concepts represented by a model will usually be at M1 or higher, modeling tools implemented with traditional languages do not allow for representing them there. Instead, they have to be located as objects as M0.



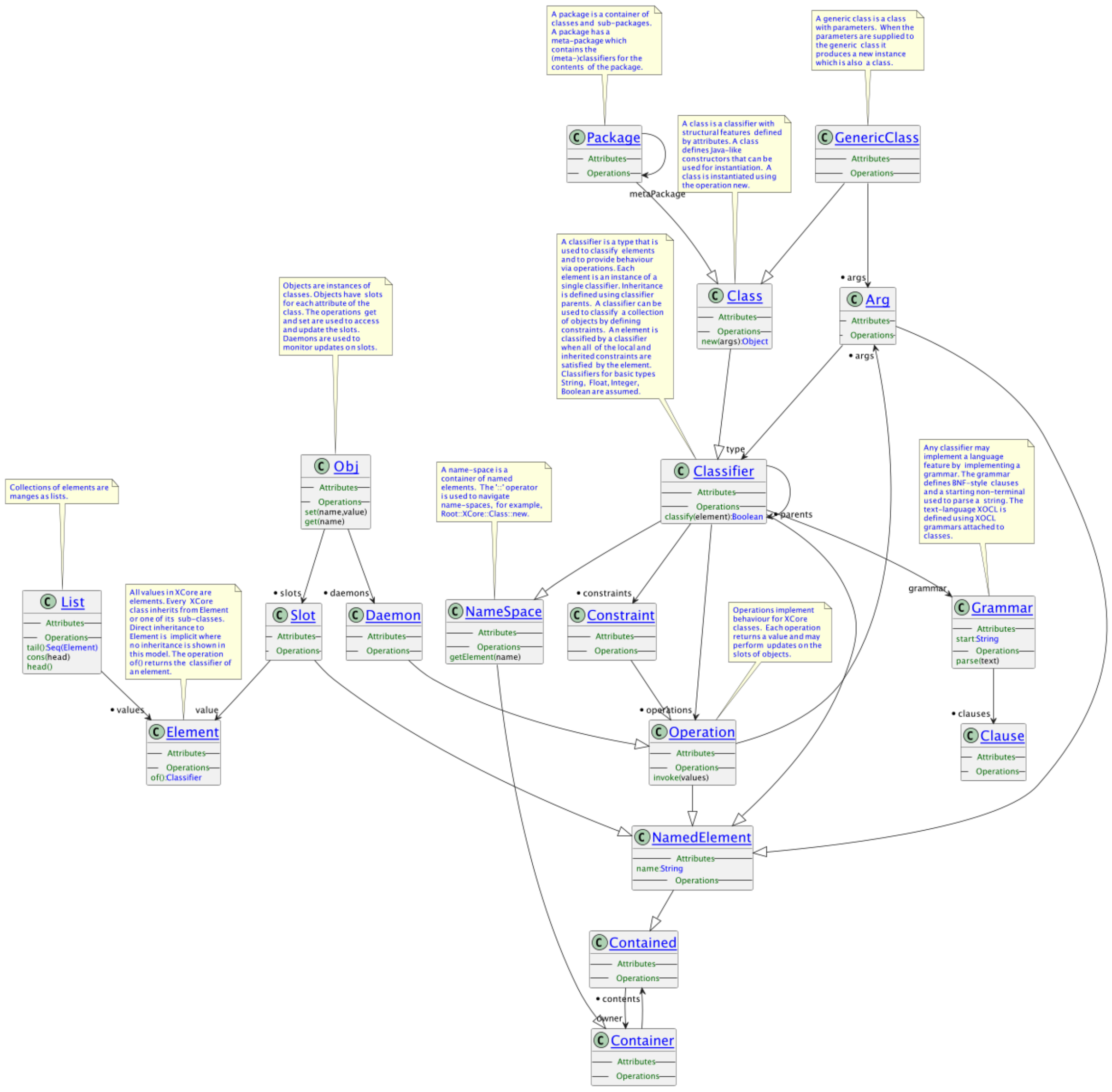
```
class Customer
{
    String firstName;
    String lastName;
    Date dateOfBirth;

    public int yearsOfAge()
        ....
}
```

Figure 2: The reason for software generation

Dissatisfactory integration of models and software: This limitation is directly related to the previous one. To empower users of a complex application system such as an ERP system, it would be useful to tightly integrate the system with the conceptual models, it is based on. In an ideal case, users could then navigate from the application they use to the corresponding models. By changing the models they could modify the software. This vision of "self-referential enterprise systems" [7] evolved already some time ago. However, its implementation failed due to limitations of prevalent programming languages that require separate representations of model and code.

The development of the XModeler^{ML} was mainly motivated by these obstacles imposed by traditional (meta) modeling and programming languages. To overcome these obstacles a language architecture was required that allows for multiple, better: an arbitrary number of classifications, and that, thus, allows for a common representation of models and code. The language architecture that builds on *XCore* and *XOCL* and that is provided by the *XModeler* (see below) proved to serve as a powerful foundation for achieving this objective.



A package is a container of classes and sub-packages. A package has a meta-package which contains the (meta-)classifiers for the contents of the package.

A generic class is a class with parameters. When the parameters are supplied to the generic class it produces a new instance which is also a class.

A class is a classifier with structural features defined by attributes. A class defines Java-like constructors that can be used for instantiation. A class is instantiated using the operation new.

A classifier is a type that is used to classify elements and to provide behaviour via operations. Each element is an instance of a single classifier. Inheritance is defined using classifier parents. A classifier can be used to classify a collection of objects by defining constraints. An element is classified by a classifier when all of the local and inherited constraints are satisfied by the element. Classifiers for basic types String, Float, Integer, Boolean are assumed.

Objects are instances of classes. Objects have slots for each attribute of the class. The operations get and set are used to access and update the slots. Daemons are used to monitor updates on slots.

A name-space is a container of named elements. The '::' operator is used to navigate name-spaces, for example, Root::XCore::Class::new.

Any classifier may implement a language feature by implementing a grammar. The grammar defines BNF-style clauses and a starting non-terminal used to parse a string. The text-language XOCL is defined using XOCL grammars attached to classes.

Collections of elements are manges as lists.

All values in XCore are elements. Every XCore class inherits from Element or one of its sub-classes. Direct inheritance to Element is implicit where no inheritance is shown in this model. The operation of() returns the classifier of an element.

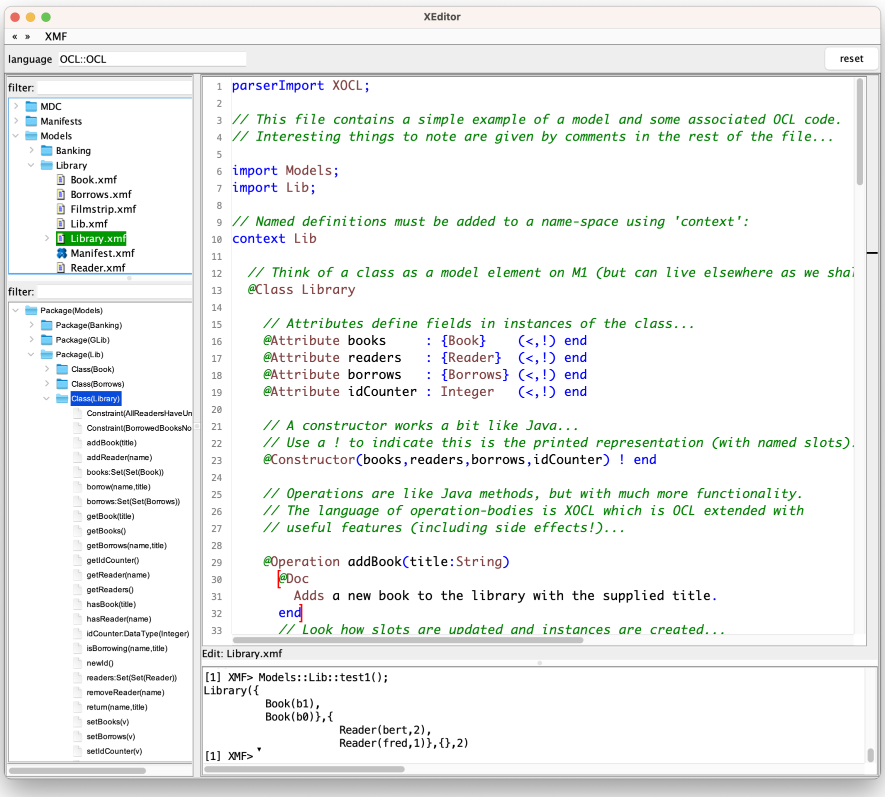
Operations implement behaviour for XCore classes. Each operation returns a value and may perform updates on the slots of objects.

Figure 3: XCore

3. Foundations

XModeler has been designed for language engineering, both in terms of model-based languages and text-based languages, and their associated tools. To achieve this it is based on XCore, which is a small meta-circular meta model that is shown in figure 3. For a simplified representation of XCore see [8]. XCore runs against a virtual machine written in Java. The two key principles of XCore are that it is both self-describing and extensible; it achieves that in the following ways: *uniformity* everything is an object with a standard interface that allows the object's representation, type and behaviour to be inspected and modified; *types* are both objects and extensible so that new types-of-types can be defined; *monitoring* updates to objects can be monitored by daemons which allows XModeler to monitor its own updates and to take appropriate action; *language support* in the form of grammars that allow new text-based language features to be incrementally added to the existing language; *meta-object-protocol* which allows the object-oriented language interface for XCore (object creation, operation invocation, slot-access and update) to be extended at the type level.

The XOCL language is implemented as a concrete language on top of XCore. Most of XModeler is written in XOCL. It is also suitable for creating executable models of systems. XEditor is a tool that is



```
parserImport XOCL;

// This file contains a simple example of a model and some associated OCL code.
// Interesting things to note are given by comments in the rest of the file...

import Models;
import Lib;

// Named definitions must be added to a name-space using 'context':
context Lib

  // Think of a class as a model element on M1 (but can live elsewhere as we sha
  @Class Library

    // Attributes define fields in instances of the class...
    @Attribute books     : {Book}    (<,!) end
    @Attribute readers   : {Reader}  (<,!) end
    @Attribute borrows   : {Borrows} (<,!) end
    @Attribute idCounter : Integer   (<,!) end

    // A constructor works a bit like Java...
    // Use a ! to indicate this is the printed representation (with named slots).
    @Constructor(books,readers,borrows,idCounter) ! end

    // Operations are like Java methods, but with much more functionality.
    // The language of operation-bodies is XOCL which is OCL extended with
    // useful features (including side effects!)...

    @Operation addBook(title:String)
      @Doc
        Adds a new book to the library with the supplied title.
      end
      // Look how slots are updated and instances are created...
```

```
[1] XMF> Models::Lib::test1();
Library({
        Book(b1),
        Book(b0)},{
                  Reader(bert,2),
                  Reader(fred,1)},{},2)
[1] XMF>
```

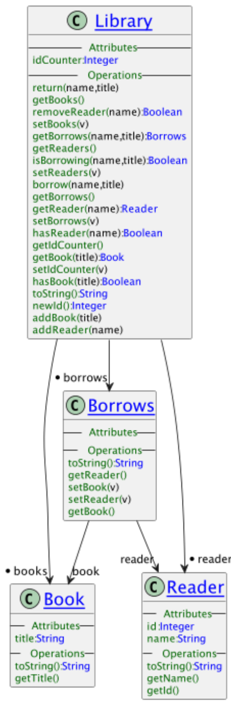


Figure 4: A Simple Executable Model in XEditor and Associated Diagram

written for working with text-based languages that are implemented in XCore. A typical executable model written in XOCL and its associated diagram is shown in figure 4.

4. Meta-language and components

The XModeler^{ML} features a multi-level (meta) modeling language, the FMML^{x} [8]. Among other things, it allows for an arbitrary number of classification levels and deferred instantiation (a property defined at level n can be instantiated at levels < n-1). Core concepts of the FMML^{x} and its default concrete syntax are shown in Fig. 5. Since the FMML^{x} is based on the golden braid architecture featured by XCore, it does not require distinguishing between a “linguistic” and an “ontological” meta model other approaches are based on [9], [10].

The XModeler^{ML} allows to overcome the obstacles imposed by traditional modeling and programming languages. It offers clearly more expressiveness by allowing for an arbitrary number of classification levels and for deferred instantiation (see, e.g., the attribute **salesPrice** of the class **Product** at level 3 in in Fig. 5). Last but not least, models created with the FMML^{x} are fully executable. The common distinction between models and code does not exist anymore, since both share the same representation.

It also enables a more efficient specification of DSMLs. The specification of a DSML within the XModeler^{ML} is illustrated in a screencast provided at the project webpages (www.le4mm.org/xmodelerml). DSMLs can be defined at any level of classification. For example, a DSML that represents general characteristics of products could be defined at level 4, and then used to specify a more specific DSML to model vehicles, which would start at level 3, but also comprise more specific concepts such as vehicle models at level 2. As a consequence, the distinction between modeling language, model and instantiations of models is overcome, since a model designed with the FMML^{x} may include classes at different classification levels. This corresponds directly to the use of concepts in natural language, where a sentence may refer to concepts at different levels of abstraction as well as to particular exemplars. For language designers, modelers and model users, this architecture creates obvious advantages. Language designers are no longer forced to develop new languages from scratch using generic concepts such as *Class* or *Attribute*.

Instead, they can use a more general, yet domain-specific language, to define a more specific one. Modelers can navigate to the language they use (it can be part of the same model) at runtime. They can also instantiate a model at runtime, which may give valuable feedback for their design decisions. Finally, this architecture gives users of an application system the opportunity to navigate to the conceptual model of the system they work with at runtime, and change the model (if they are authorized to do so). By changing the model, they would directly change the system.

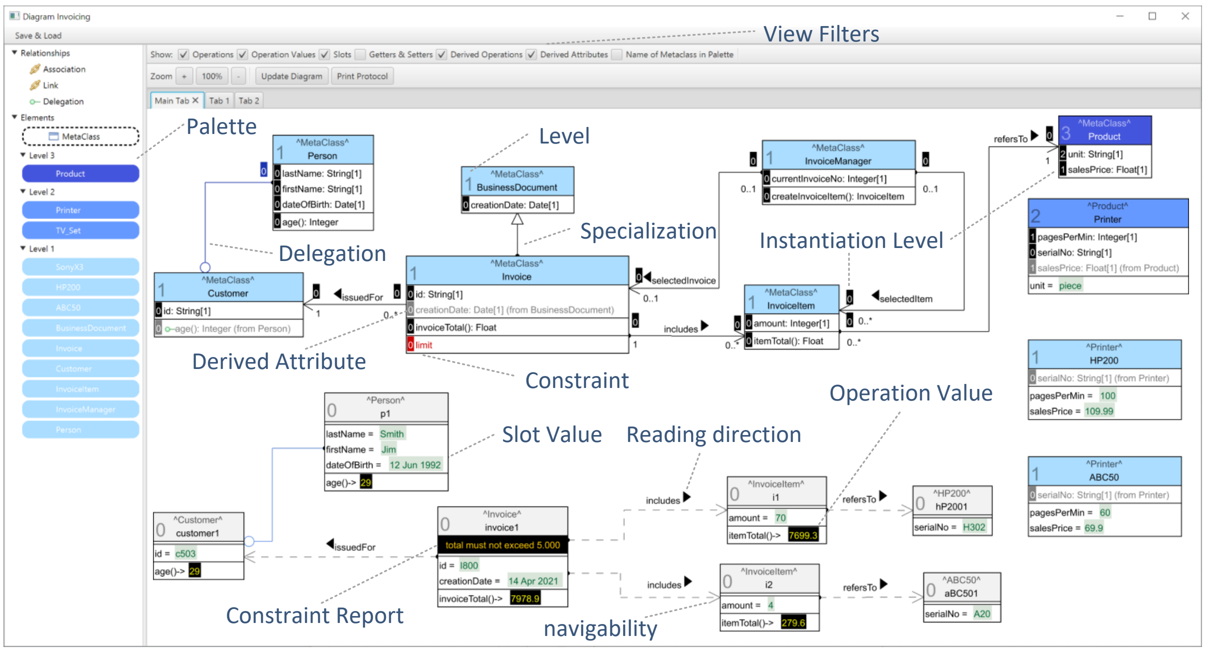


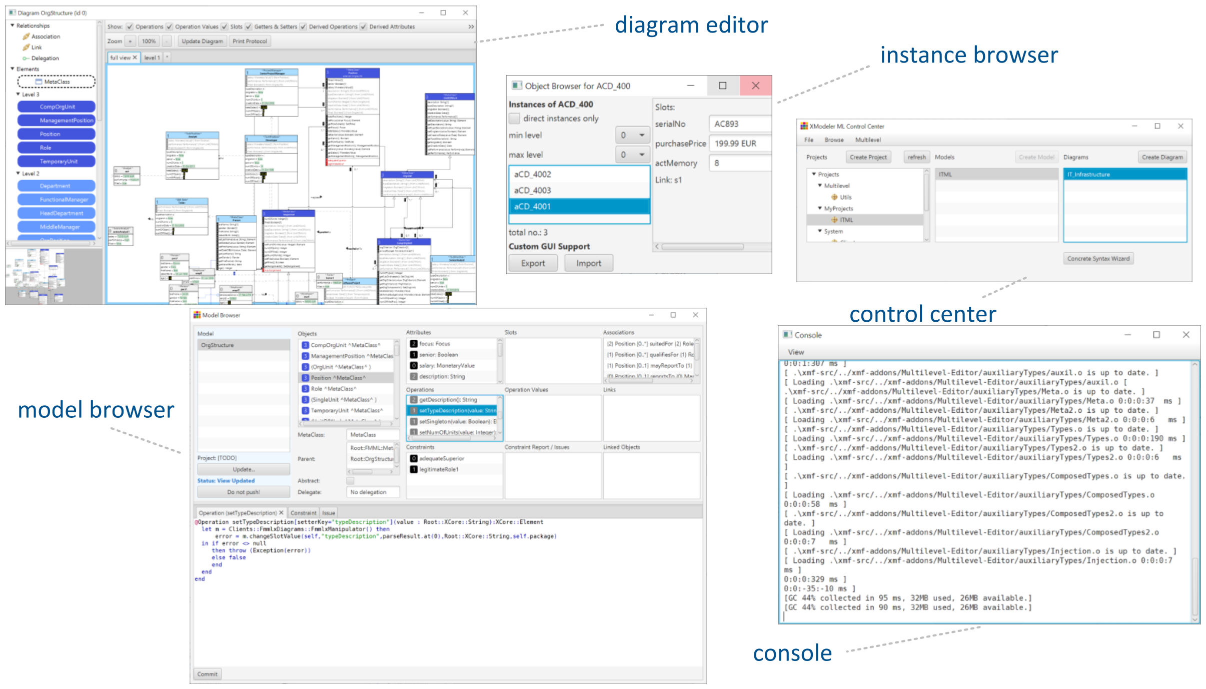
Figure 5: Illustration of concepts provided by the FMMLˣ

With the XModeler$^{\mathrm{ML}}$, the notorious synchronization of models and code is widely obsolete. All classes specified with the FMMLˣ are specified and implemented (!) at the level where they conceptually belong. Hence, they can be instantiated within a model editor without the need for code generation.

The XModeler$^{\mathrm{ML}}$ consists of various components that allow accessing, manipulating and executing a model from different perspectives. With each model a corresponding *diagram editor* is generated. In addition to allow for creating and modifying models, it also allows for executing models. If the execution of a model results in changes of the diagram, these are directly shown in the editor.

Developers that prefer textual representations over graphical ones may use an instance of the *model browser*. It can operate on the same model that is also used by an instance of the diagram editor at the same time. In case the default notation of the FMMLˣ (see Fig. 5) is not satisfactory, the *concrete syntax editor* serves the specification of notations by arranging pre-defined SVGs and text elements. After a notation has been defined, it can directly be used by the diagram editor.

The *workspace* allows to interact with and manipulate model elements, e.g. for testing purposes, using a command line editor. Two components serve the realization of application style GUIs to interact with models. An *object browser* can be generated for every class in a model. It is composed of widgets such as text or list boxes that serve the presentation of instances of the selected class. The GUIs created by the object browser are restricted to objects of one class and lack a user-friendly layout of widgets. The *GUI builder* overcomes this restriction. It transparently sends a model to an external GUI builder, which allows to modify a generated GUI. After the GUI was redesigned, it is transparently sent back to the XModeler$^{\mathrm{ML}}$, where it can be used immediately to interact with the corresonding model.



```
@Operation setTypeDescription[setterKey="typeDescription"](value : Root::XCore::String):XCore::Element
  let m = Clients::FmmlxDiagrams::FmmlxManipulator() then
      error = m.changeSlotValue(self,"typeDescription",parseResult.at(0),Root::XCore::String,self.package)
  in if error <> null
     then throw (Exception(error))
     else false
     end
  end
end
```

```
0:0:1:307 ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/auxil.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/auxil.o [
.\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Meta.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Meta.o 0:0:0:37  ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Meta2.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Meta2.o 0:0:0:6   ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Types.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Types.o 0:0:0:190 ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Types2.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Types2.o 0:0:0:6   ms
]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/ComposedTypes.o is up to date.
]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/ComposedTypes.o
0:0:0:58  ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/ComposedTypes2.o is up to
date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/ComposedTypes2.o
0:0:0:7   ms ]
[ .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Injection.o is up to date. ]
[ Loading .\xmf-src/../xmf-addons/Multilevel-Editor/auxiliaryTypes/Injection.o 0:0:0:7
ms ]
0:0:0:329 ms ]
0:0:-35:-10 ms ]
[GC 44% collected in 95 ms, 32MB used, 26MB available.]
[GC 44% collected in 90 ms, 32MB used, 26MB available.]
```

Figure 6: Components of the XModeler^{ML}

Fig. 6 shows examples of the diagram editor, the model browser, the workspace and the object browser. It also shows the *control center*, which, among other things, serves to load models, to start diagram editors or access the workspace.

The project webpages at www.le4mm.org offer a wide range of resources on the XModeler^{ML}, including publications, screencasts and downloads of the latest build as well as example models. Multi-level modeling represents a new language paradigm. As a consequence, using the FMML^{x} may be perceived as demanding at first. Therefore, it will often be a good idea to start with using the FMML^{x} as a kind of UML class diagram editor. One would then create a UML class diagram which could be directly instantiated, and executed, within the diagram editor (see screencast at www.le4mm.org/xmodelerml/#UML-pp). Subsequently, one could continue with creating a first multi-level model to then subsequently raise the level of abstraction (screencast at www.le4mm.org/xmodelerml).

5. Conclusions and future work

Research on multi-level modeling has resulted in various languages and tools, e.g., [11, 12, 9, 13]. To the best of our knowledge, the XModeler^{ML} is the only multi-level modeling environment that allows for the execution of models at all levels of classification. A more detailed comparison of the XModeler^{ML}with other tools would go beyond the scope of this paper. For an elaborate comparison with the LML and Melanee see [14].

In conjunction with the language engineering facitilities provided by XModeler and XOCL, it enables a powerful foundation not only for the development of modeling languages and model editors, but also for a new generation of self-reflexive application systems. While still in the state of a research prototype, we believe the XModeler^{ML} allows to experience the benefits of multi-level modeling and software development in a fairly convenient way.

At the same time, multi-level modeling is a research subject that offers attractive perspectives. These include the development of hierarchies of DSMLs where higher level DSMLs promise attractive economies of scale where more specific DSMLs serve to address particular needs [6]. The fact that the XModeler^{ML} allows for executing models also enables attractive options for providing end-users with tools to develop or modify small applications that go clearly beyond the potential of current "low-code" environments. Furthermore, the technology underneath the XModeler^{ML} is especially suited for the design and implementation of “digital twins”. To manage digital twins, it is not sufficient to model properties of their types. Furthermore, it is important to account for the state of particuar instances, too. That requires to also to represent knowledge about specific characteristics of instances and to manage them during execution. With traditional modeling and programming languages this is hardly possible without extensive workarounds. In contrast, multi-level language architectures provide the concepts for a straightforward implementation of digital twins.

References

[1] U. Frank, T. Clark, Language Engineering for Multi-Level Modeling (LE4MM): A Long-Term Project to Promote the Integrated Development of Languages, Models and Code, in: J. Font, L. Arcega, J.-F. Reyes-Román, G. Giachetti (Eds.), Proceedings of the Research Projects Exhibition at the 35th International Conference on Advanced Information Systems Engineering (CAiSE 2023), CEUR, 2023, pp. 97–104.

[2] T. Clark, P. Sammut, J. Willans, Applied Metamodelling: A Foundation for Language Driven Development, 2 ed., Ceteva, 2008.

[3] T. Clark, P. Sammut, J. S. Willans, Super-languages: Developing languages and applications with XMF (second edition), CoRR abs/1506.03363 (2015). URL: http://arxiv.org/abs/1506.03363. arXiv:1506.03363.

[4] U. Frank, Multiperspektivische Unternehmensmodellierung: Theoretischer Hintergrund und Entwurf einer objektorientierten Entwicklungsumgebung, Oldenbourg, München, 1994.

[5] J. Gulden, U. Frank, MEMOCenterNG – A full-featured modeling environment for organisation modeling and model-driven software development, in: Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development (FTMDD 2010), 2010.

[6] U. Frank, Multi-Level Modeling: Cornerstones of a Rationale, Software and Systems Modeling 21 (2022) 451–480.

[7] U. Frank, S. Strecker, Beyond ERP Systems: An Outline of Self-Referential Enterprise Systems, No. 31, University of Duisburg-Essen, 2009, 2009.

[8] U. Frank, The Flexible Multi-Level Modelling and Execution Language (FMML^{x}). ICB Research Report, No. 66, University of Duisburg-Essen, 2018 , 2018.

[9] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, Software & Systems Modeling 7 (2008) 345–359.

[10] C. Atkinson, B. Kennel, B. Goß, The level-agnostic modeling language, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering, volume 6563 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2011, pp. 266–275.

[11] B. Neumayr, K. Grün, M. Schrefl, Multi-level Domain Modeling with M-objects and M-relationships, in: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96, APCCM ’09, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2009, pp. 107–116.

[12] J. de Lara, E. Guerra, Deep meta-modelling with metadepth, in: J. Vitek (Ed.), Objects, Models, Components, Patterns, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 1–20.

[13] M. A. Jeusfeld, B. Neumayr, DeepTelos: Multi-level Modeling with Most General Instances, in: I. Comyn-Wattiau, K. Tanaka, I.-Y. Song, S. Yamamoto, M. Saeki (Eds.), Proceedings of the 35th International Conference on Conceptual Modeling (ER 2016), Springer, Cham, 2016, pp. 198–211.

[14] A. Lange, U. Frank, C. Atkinson, D. Töpel, Comparing LML and FMML^{x}, in: ACM/IEEE (Ed.), Proceedings of the International Conference on Model Driven Engineering Languages and Systems, IEEE Conference Publishing Services, Los Alamitos, CA, Washington, Tokyo, 2023.