A Domain-specific Language for Modeling Method Definition: from Requirements to Grammar

Niksa Visic
Faculty of Computer Science,
University of Vienna,
Vienna, Austria,
niksa.visic@univie.ac.at

Hans-Georg Fill
Faculty of Computer Science,
University of Vienna,
Vienna, Austria,
hg@dke.univie.ac.at

Robert Andrei Buchmann
Faculty of Economic Sciences and Business Administration,
Babes-Bolyai University,
Cluj Napoca, Romania,
robert.buchmann@econ.ubbcluj.ro

Dimitris Karagiannis
Faculty of Computer Science,
University of Vienna,
Vienna, Austria
dk@dke.univie.ac.at

***Abstract*— The core process a modeling method engineer needs to accomplish starts with the acquisition of domain knowledge and requirements, and ends with the deployment of a usable modeling tool. In between, a key intermediate deliverable of this process is the modeling method specification which, ideally, should be platform independent. On one hand, it takes input from a structured understanding of the application domain and scenarios; on the other hand, it provides sufficiently structured input to support the implementation of tool support for modeling activities. It is quite common that such modeling methods are domain-specific, in the sense that they provide concepts from the domain as "first-class modeling citizens". However, for the purposes of this paper, we raise the level of abstraction for "domain specificity" and consider "modeling method engineering" as the application domain. Consequently, we raise several research questions - whether a domain-specific language can support this domain, and what would be its requirements, properties, constructs and grammar. We propose an initial draft of such a language – one that abstracts away from meta-modeling platforms by establishing a meta^2 layer of abstraction where a modeling method can be defined in a declarative manner, then the final modeling tool is generated by automated compilation of the method definition for the meta-modeling environment of choice.**

Keywords*— *domain-specific language; modeling method; meta-modeling; modeling tool

I. INTRODUCTION

The goal of this paper is to introduce a domain-specific language (DSL) that supports the realization of modeling methods by establishing a meta-meta layer that abstracts away from common meta-modeling environments, allowing the method engineer to focus on the conceptual building blocks of a modeling method rather than on a meta-modeling platform's technical specificity. A demonstrative editor and compiler have been implemented to evaluate feasibility of the proposal.

The work is grounded in the notion of a **modeling method**, as introduced by [1] and further refined by [2]. Therefore, for the purposes of this paper, a modeling method is defined in terms of several building blocks – modeling language, procedure and functionality (mechanisms and algorithms for model processing) - which provide "first-class citizen" concepts for the hereby proposed domain-specific language (the relation between these building blocks and the DSL constructs will be discussed in Section III). In other words, *modeling method engineering* becomes the application domain addressed by the "domain specificity" of the introduced language. This is the sense in which we will use "domain-specific" and the DSL acronym throughout the entire paper (not to be confused with the domain specificity of modeling methods that can be created with the proposed language).

Relative to OMG's MOF framework [3], the language provides a meta^2-model that supports the compilation of a modeling method definition (in terms of the proposed DSL) for deployment on a meta-modeling environment of choice (thus producing a modeling prototype for end-users).

The work is complemented by the contextual goal of advocating the notion of *agile modeling method engineering* (within the methodological context of the Open Model Initiative Laboratory [4]), inspired by principles of agile software engineering. The main assumption is that a modeling method is not necessarily fixed (or driven by the versioning of standard languages like UML [5] or BPMN [6]). Instead, modeling methods may evolve iteratively based on changing modeling requirements and feedback loops. The DSL introduced here is aimed to support a quick text-based declarative editing of a modeling method definition, which can be compiled for fast prototyping on the meta-modeling environment of choice. An *agile modeling method* distinguishes itself from established standards (e.g. UML, BPMN) by several characteristics:

1. *Deeper specialization* driven by the end-user's application domain to the detriment of reusability across domains. There is always a trade-off in the design of modeling languages between reusability and requirements coverage. Agility is a key requirement for methods aiming to serve a narrow domain, or even the internal needs of an enterprise;

2. *Richer semantics* captured both on notational level (as visual cues in the language symbols) and on semantic level (as editable properties). Semantics are, again, driven by requirements which may come directly from modeling stakeholders, or indirectly from required functionality (e.g.

978-1-4673-6630-4/15/$31.00 ©2015 IEEE

model queries, run-time applications that must consume models etc.). Typically they are extension or hybridization requirements (e.g. the extension of i* into DEST [7]);

3. *Granular evolution driven by changing requirements.* Standards are more rigid in this respect, with a versioning process determined by standardization bodies and broad requirements consolidated from a global community. Just as in agile software engineering, requirements may change due to multiple factors – improved common understanding of the domain, additional capabilities requested by end-users, additional semantics required as input by various model-driven runtime components, or even a collateral effect of some organizational evolution in the sense discussed by [8].

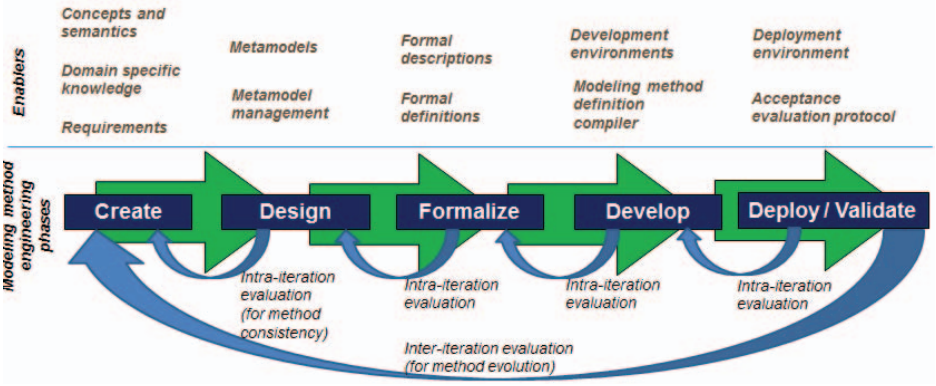

Fig. 1. The sequential structure of one iteration during agile modeling method engineering

Establishing a full methodology of agile method engineering is not in the scope of this paper – however its inherent iterative nature provides motivational rationale for positioning the work. The overall process for realizing one iteration of an evolving modeling method (including its modeling language) encompasses several phases (Fig. 1), according to the framework established by the Open Model Initiative Laboratory [4]:– (1) create, (2) design, (3) formalize, (4) develop, and (5) deploy/validate. Inter-phase validation may occur within the same iteration, to ensure consistency between intermediate deliverables. The process reiterates as knowledge emerges gradually from the domain, together with evolving modeling requirements.

The ***creation*** phase is a mix of knowledge acquisition and requirements elicitation processes, with the aim of defining the *modeling language requirements* (concepts and relations needed in a modeling language) and the *modeling functionality requirements* (e.g. competence questions that models should be able to answer; decisions to be supported by model analysis; other functionality pertaining to model visualization, checking or transformation). This phase may also benefit from analyzing requirements pertaining to the run-time systems that will consume the models, if interoperability between design-time and run-time is needed. Different types of knowledge are sought for – e.g. procedural (processes), motivational (goals), relational (dependencies, constraints) as well as the semantics of first-class entities to be included. Hence a mix of techniques ranging from documentation analysis to direct stakeholder interviewing must ensure a common understanding between the methodologist and those who will apply the method.

The ***design*** phase is the core meta-modeling effort for specifying a structured metamodel, the language grammar, the recommended visualization and functionality. Existing languages commonly used for domain modeling (e.g. class diagrams, ER diagrams) may be used for this purpose, although dedicated languages can be tailored to include metamodel management functionality (e.g. tracking metamodel changes). The DSL introduced by this paper takes direct input from this phase, as the metamodel design specification becomes executable once translated in the proposed DLS's constructs.

The ***formalization*** phase describes the outcome of previous phases non-ambiguously, either with the purpose of sharing them within a scientific community (in terms of algebra, first order logic, Petri Nets etc.) or with the purpose of preparing the method implementation (in terms of the hereby introduced DSL). A formalization framework has been established in [9] to describe metamodels and models for the ADOxx meta-modeling platform [10].

The ***development*** phase involves a concrete meta-modeling platform in order to produce a modeling prototype, as well as a compiler that translates the hereby introduced DSL abstraction to the technology-specific constructs of the targeted platform.

The ***deployment/validation*** phase deals with packaging and installing the modeling prototype for user acceptance tests. The deployment may consider different options such as standalone modeling tools or cloud-enabled modeling-as-a-service. The feedback and lessons learned feed into the next iteration, together with any possible additional requirements which normally emerge from first-hand experience of potential users and a gradual understanding of how a modeling method can support them.

The DSL proposed in this paper aims to support the design-to-development phases and enables a quick redeployment of multiple iterations for an evolving modeling method and across multiple meta-modeling platforms. For demonstration purposes a proof-of-concept editor and a compiler for the ADOxx meta-modeling platform are showcased.

The paper will present the DSL in terms of three facets of the **research challenge** – (a) the motivating requirements, (b) formal grammar aspects and (c) details on the proof-of-concept implementation. The remainder of the paper is organized as follows: Section II will briefly cover the state of the art in

language design, which was the baseline for developing the DSL. Section III provides an overview of various classes of requirements that have been considered in the proposed DSL's development, based on an in-depth analysis of various meta-modeling platforms and a repository of modeling methods accumulated through the Open Model Initiative. Section IV provides excerpts from the proposed language's EBNF grammar specification, while Section V describes a minimal running example serving as "validation by instantiation" of the designed artefact. The paper concludes with a SWOT evaluation and formulates a takeaway message.

II. Related Work

The state of the art in language design, especially in the area of domain-specific languages and modeling languages has been extended in the recent years, with concepts showing significant growth in their maturity both in academic research and in industrial use [11]. Some of the notions involved in the design of domain-specific languages provide means of defining custom programming/modeling artifacts or code generation facilities. The most relevant of these are further discussed below:

Language Oriented Programming [12] is a novel way of organizing the development of a large software system. The approach starts by developing a formally specified, domain-oriented, high-level language which is well-suited to develop the system under consideration. After the system has been implemented in the before developed language, it is translated using a compiler or an interpreter to existing technology. Among claimed advantages for domain analysis, rapid prototyping, maintenance, portability, and reuse of development work, LOP provides higher development productivity and faster time to market.

Language Driven Development [13] is a software development method which involves the use of multiple DSLs at various points in the development life-cycle. It is based on the ability to rapidly design new languages and tools in a unified and interoperable manner. By allowing engineers and domain expert to express their designs in the language that they are most comfortable with and that will give them most expressive power, productivity can be increased. The LDD vision relies heavily on the language integration. Languages should be weaved together to form a unified view of the software system.

Model Driven Architecture [14] is a term commonly used for the generation of program code from (semi-)formal models (e.g., UML, UML profiles, various DSLs). System functionality is defined using a platform-independent model (PIM) which is described in an appropriate DSL. The PIM is then translated to one or more platform-specific models (PSM) that computers can run. The transformation process is generally automated by dedicated tools [15].

Software Factories [16] refer to software assets used to create specific types of software components. They help structure the development process and are used for developing languages that support the construction of software components. A software factory may include processes, templates, integrated development environment (IDE) configurations and views. The type of software a factory may produce is defined when the factory is created.

Superlanguages [17] provide control over all aspects of representation and execution. They can be extended with new features, which can be seamlessly weaved into the existing features. Execution mechanisms can be changed to reflect the needs of each new application. Superlanguages also provide a powerful control over the language engine via meta-features – a way of tailoring a language in a modular way without polluting programs with unnecessary code.

"Language Workbench" is a term proposed by Martin Fowler to designate the IDE support for development of domain-specific languages [18][19]. There are many examples of such IDEs, some of them specifically designed for development of textual languages (e.g., Xtext [20], Irony [21]), and some for development of graphical languages (e.g., VS Visualization & Modeling Tools [22], MetaEdit+ [23]).

"Meta-modeling Platform" [1] is a term used to describe an environment specifically targeting the development of graphical modeling languages and modeling methods using a meta-modeling approach – a layered approach (see OMG's MOF [3]) where one describes the modeling language structure by instantiating an already existing meta^{2}model provided by the platform. Because of this approach, a platform can provide support to the modeling language being developed through already existing features and functionality (e.g., algorithms and mechanism for model analysis and simulation) [24].

Closely related works are Graphiti (Eclipse-based) [25] and XModeler/XMF [26], in the sense that they also provide a Language Oriented Programming approach to metamodeling. However, they do not aim to provide a new layer of abstraction relative to the variety of existing platforms (via platform-specific compilers); instead they are themselves standalone meta-modeling platforms with some significant productivity improvements.

The work at hand relates to the notion of Meta-modeling Platform in the sense that it abstracts away from its instances while providing compilers to transfer a modeling method definition to a specific meta-modeling technology. The concept makes use of lessons learned from Language Oriented Programming and Language Driven Development, by transferring some of their principles to the discipline of meta-modeling. Its implementation can be considered itself a Language Workbench for the development of modeling languages, therefore the paper makes to the domain of modeling method engineering a contribution inspired by meta-programming experiences.

III. Classes of Requirements for a DSL for Modeling Method Definitions

The *primary class of requirements* comes from the language's application domain. As mentioned previously, the application domain for the proposed DSL is modeling method engineering. The *secondary requirements* come from the meta-modeling platforms that act as candidates for deployment environments, therefore providing dedicated functionality that can be rather generic (e.g. the way of defining a metamodel) or

more specific (e.g. the way of defining mechanisms for model analysis and simulation). The *tertiary requirements* come from generally accepted principles and best practices governing the overall design process of programming and domain-specific languages, including the definition of language statements, control structures, expressions etc. The *quaternary requirements* come from the outlook on the emerging and future technologies and their possible influence on modeling and meta-modeling techniques. To accommodate emerging approaches (e.g., models, methods, and modeling tools as a service) the DSL under consideration should support its own evolution through language extensibility (i.e. introduction of new domain concepts) and metamorphosis (i.e. complete syntax change). These classes of requirements will be further analyzed in this section.

A. The Application Domain: Modeling Method Engineering

The design of a DSL for modeling methods entails a very specific set of domain-specific features. To understand the requirements of such a language one needs to get familiar with various modeling method characteristics and isolate the most important ones.

The **modeling method** building blocks are adapted here from the work of Karagiannis and Kühn (see [1][24]) with a slight deviation due to the formalization possibilities (Fig. 2): (1) a modeling language, (2) modeling algorithms, and (3) mechanisms and modeling procedures.

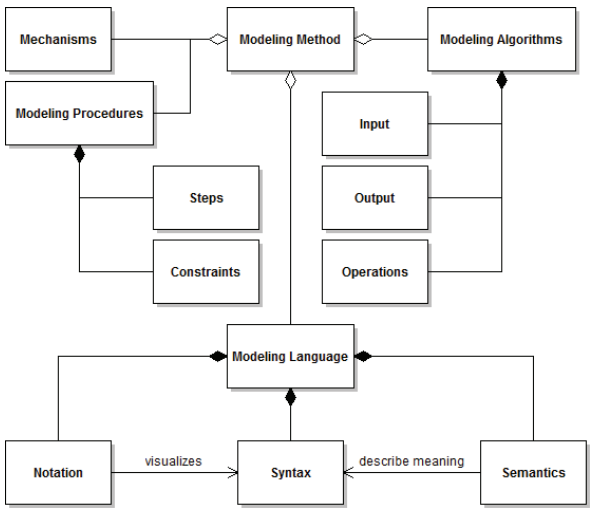

Fig. 2. The building blocks of a modeling method

The primary building block is the graphical **modeling language**, further structured in (abstract) *syntax, semantics* and *notation*. The *syntax* provides the language grammar, typically through production or well-formedness rules. The *semantics* give meaning to the syntax of a language, defining the terminological taxonomy of symbols, the property set for each class of syntactical constructs, including their relations and semantic constraints that must be applied to them. The *notation* defines the graphical symbols of a modeling language together with the morphological variations dictated by semantics (visual variations determined dynamically by the values of some key properties). a graphical representation of a modeling language. Not all constructs introduced in the syntax of a modeling language need to have a graphical representation – these constructs are typically considered abstract (i.e. not to be instantiated in models) and are present on the higher layers of the terminological taxonomy, typically for reusability of semantics (e.g. property inheritance).

The secondary building block comprises **modeling algorithms**. These define model-processing functionality built on top of the language structure (e.g. model analysis, simulation, evaluation).

The tertiary building block comprises aspects that are not covered by the hereby proposed DSL. On one hand, **mechanisms** represent functionality that is native to the meta-modeling platform used for implementation and deployment, therefore technology-specific (e.g. model publishing services). On the other hand, the modeling procedure is typically a methodological component, expressed through informal guidelines and steps that need to be taken by modelers to reach their goal. These can be complemented by modeling algorithms providing automated validation and checks for particular steps (e.g. to block the creation of swimlanes in a business process model unless an organizational model was previously defined to provide semantics to those swimlanes).

The DLS for modeling method definition draws its main class of requirements from these building blocks.

B. Analyzing the Concepts from the Application Domain

To understand what kind of first-class constructs are required in a modeling method definition DSL, various existing modeling languages have been scrutinized. For example, BPMN has four distinct groups of constructs: flow objects (event, activity, and gateway), connecting objects (sequence flow, message flow, and association), swim lanes (pool and lane), and artifacts (data object, group, annotation). Petri Nets [27] consist of places, transitions, and arcs. UML class diagram structures a model using classes, their attributes, operations and relationships among the classes. Finite state machine diagrams [28] used to design both computer programs and sequential logic circuits employ constructs like state and transition.

A closer look reveals that all of these modeling languages rely on the common notions of knowledge representation: a concept of a *class* (e.g. place, class, event, activity, state), a concept of *relationship* having classes as domain and range (e.g., sequence flow, association, arcs, transitions), and a concept of *attribute* that specifies class semantics (e.g. name, cost, color, height, position, salary). This has been addressed in the past many times as meta-elements by the OMG's MOF but can be traced back to Parmenide's way of describing reality in terms of categories of being and provides the base for knowledge representation approaches. Ultimately the metamodel of a modeling language is an ontological view on the domain addressed by that language. There exist many meta^{2}models providing similar primitive meta-elements (Ecore, GME, GOPPRR and ADOxx – see the overview of [29]) with slightly different naming conventions (property instead of attribute, atom instead of class, etc.) or with higher level

constructs (roles on properties can be considered relations of higher arity) – however they can be reduced to these basic notions [29] which, therefore, must also be captured in a DSL for modeling method definition.

The notions of class, relationship, attribute as well as different ways of relating them (domain, range, specialization) are the main tools of *abstraction*. However, besides abstraction, a modeling method also requires *decomposition*, which is necessary to manage the complexity of models. Decomposition can be present (a) *as a language construct* in the models themselves (as a grouping container e.g. a swimlane in business processes, or as a link between an element and its parts, e.g. a business process and its subprocesses), but can also be (b) *established at the method definition level*, by partitioning the language syntax into *model types* (groups of constructs that can address particular problems). Having relationships defined across model types will ensure that they can be structurally and semantically connected.

Using the four notions – class, relationship, attribute, and model type – a typical modeling language syntax (how modeling objects can be connected by modeling relations) and semantics (property sets of classes, model types and relations) can be defined. Additional constructs in the DSL are needed to the extent of desired expressivity for constraints and rules. For this, the extensibility of the DSL must be considered as a key requirement, not unlike the way ontology languages (e.g. OWL [30]) evolve by incorporating new built-in concepts (e.g. transitive property types, qualified cardinality restrictions).

For the current proof-of-concept implementation of the proposed DSL only the expressivity that can be directly mapped on the ADOxx meta-modeling functionality has been included (e.g. cardinality constraints).

Another requirement comes from the necessity to define the notation and its morphology (dynamic notation, mapping multiple notations to the same class, with variations dictated by its instance-level attributes).

With respect to modeling algorithms, analysis and simulation are the most common functionality, but there can be many specific scenarios for various modeling methods (see [31] for more information). The basic building blocks for describing an algorithm can be borrowed from algorithm design and implementation in software engineering [32][33]. Those are input, operations, and output. A major part of operations are control structures: conditionals (if, else …), iterations (also known as loops: while, for …), and selective structures (switch …), partitioning code into functions.

C. Analyzing Existing Artefacts from the Domain

For further requirements analysis towards the design of a DSL for modeling methods, with a purpose of extracting the key modeling language constructs, nineteen implementations of modeling methods (developed within OMiLAB [4] and hosted on the Open Model Initiative repository of modeling method implementations [31]) have been analyzed: BEN, BIM, CIDOC (based on [34]), eduWEAVER, EKD, HORUS, IMP2.0, *i** (based on [35]), OM*i**T, InSeMeMo, MeLCa, OKM, Secure Tropos (based on [36]), UML, PetriNets, MoSeS4eGov, PROMOTE, SeMFIS (based on [37]), and VLML (based on [38]). All methods are implemented on the ADOxx metamodeling platform, together with a dozen of custom made services [39] that enhance the modeling tool development process on various levels – from the creation of graphical syntax to the generation of documentation.

After a quantitative analysis of nineteen modeling methods (Fig. 3) one can notice the diversity in the number of basic artifacts. This hints to the complexity of abstractions used to describe a specific modeling domain, with the higher numbers representing modeling methods with deeper domain specificity. While some of the methods are quite fixed due to their standard nature (e.g. UML, i*), others reclaim the requirement for agile evolution.

During the agile evolution of such a method, intuitively the number of artifacts grows as more specialization is added to the language (i.e. subtyping the artefacts). However, it can also happen that the number reduces, as it can be seen in the case of CIDOC, where the newer implementation has significantly less classes and relations. The complexity of CIDOC was transferred from classes and relations to attributes, thus improving usability (less constructs on the modeling toolbar, more configuration in the properties of those constructs)

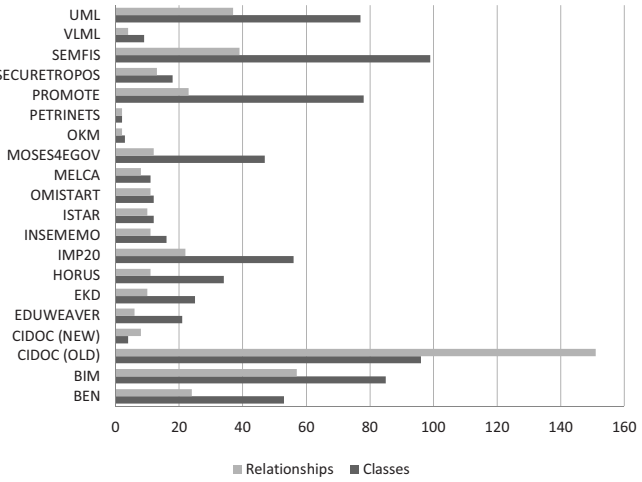

Fig. 3. Number of relations and classes in OMI implementations

During the initial requirements analysis a relatively long list of concepts which repeatedly appear in many of the analyzed modeling methods was compiled – the recurring top 10 classes in descending order are: *container*, *actor*, *label*, *resource*, *decision*, *activity*, *process*, *start*, *end*, and *goal*; the recurring top 10 relationships in descending order are: *associates*, *depends*, *flows*, *specializes*, *has*, *part of*, *contributes*, *decomposes*, *relates*, and *uses*. The mentioned concepts and their percentage of occurrence in the analyzed method definitions are indicated in Table I.

A key question that can be raised here is if such recurring concepts should be part of a domain-specific language for modeling method definition. Technically, they are on a different level of abstraction than the previously discussed notion (class, attribute), so they belong rather to the application output of the DSL rather than being first-class citizens of the DSL.

TABLE I. TOP 10 MODELING METHOD CONCEPTS AND RELATIONSHIPS WITHIN OMI

Classes		Relationships	
Name	***%***	***Name***	***%***
Container	68	Associates	42
Actor	53	Depends	37
Label	53	Flows	37
Resource	53	Specializes	37
Decision	47	Has	32
Activity	37	Part of	32
Process	37	Contributes	21
Start	37	Decomposes	21
End	37	Relates	21
Goal	32	Uses	16

However, programming languages also show this feature of extending the language with predefined instances in the form of libraries, macros etc. with the goal of providing ready-to-use application-level constructs, thus increasing productivity. Standard libraries are typically deployed together with basic programming language compilers. This can be a valuable requirement for the hereby proposed DSL, in order to provide ready-to-use modeling language fragments in the form of a "standard library", which can take the form of a predefined collection of classes, relationships, even model types.

D. Analyzing Meta-modeling Platform Functionality

The output of a DSL for modeling method definitions must be ultimately executed on a meta-modeling environment that can produce a usable modeling tool, therefore existing platforms have been analyzed to identify their functional and non-functional requirements. As a result several key meta-modeling platform components were isolated. The key conceptual components are depicted in Fig. 4 and can be mapped to functional requirements.

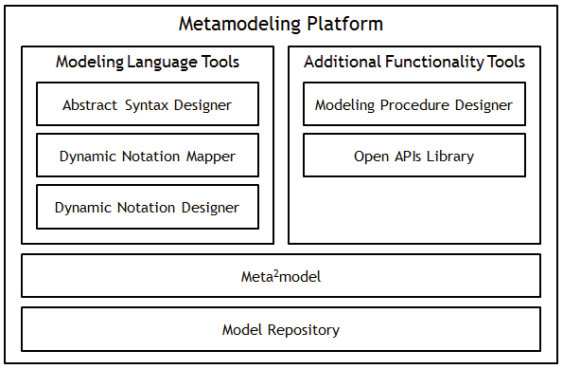

Fig. 4. Conceptual view on the components of a meta-modeling platform

1. The meta^{2}model is the key enabler from which all metamodels are instantiated, providing the core concepts and functionality. A key requirement for a meta-modeling platform is to have a meta^{2}model that is generic enough to be able to instantiate concepts from a wide range of domains, but at the same time to be sufficiently rich to enable a detailed modeling language specification;
2. A sophisticated control mechanism enabling structured abstract syntax and semantics definition and manipulation for the concepts instantiated from the meta^{2}model is the second functional requirement;
3. The development of graphical modeling languages is more complex than the development of textual languages. Textual languages (e.g., textual programming languages, textual specification languages) can have their syntax specified in a textual form (e.g., EBNF), where, most of the times, abstract and concrete syntax are joined together and defined at the same place. In case of graphical modeling languages, it is common to have multiple notations connected with one modeling element, meaning that abstract syntax can have multiple concrete syntax representations. The third functional requirement is to provide a mapping mechanism between the abstract syntax and the graphical representation;
4. Graphical modeling elements are not only static figures on the modeling canvas. They also provide a complex interface between the user and the model enabling predefined functionality (e.g., triggers for functionality, hyperlinks towards other models). Being able to design appropriate graphical representations including an embedded user interface embedded into modeling elements is a fourth requirement;
5. Modeling procedures enforce the order in which modeling elements needs to be used. In most cases this is not necessary, because one wants to give more freedom to the modeler. However, there exist modeling methods which strictly define the order one can use the modeling elements – e.g. if one wants to model information security, one should describe physical security (e.g., server locked in protected environment) before virtual security (e.g., firewall, access control, etc.). The enforcement of modeling procedures is another meta-modeling platform functional requirement;
6. Algorithms are the means which are used to define and implement additional functionality of a modeling method. To be able to use and reuse already present platform functionality for defining various algorithms and mechanism, one needs an interface to this functionality, typically realized as a well-documented and interoperable set of APIs;
7. A meta-modeling platform needs a dedicated repository for storing a modeling method definition, and for storing models defined by a modeling method. Repositories provide the possibility to reuse already defined modeling elements, to track changes for both development of modeling methods and models, to propagate changes done on the modeling method layer to the model layer.

Tot this list we add several non-functional requirements that typically characterize meta-modeling platforms:

Extensibility [40] takes under consideration future growth. It is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can mean addition of new functionality or modification of existing functionality. As complex software system, meta-modeling platforms should have a public application programming interface (API) that allows extension and modification of the platform's behavior by developers who do not have access to the original source code.

Interoperability [41] is the ability of systems to work together by exchanging information and using the information that has been exchanged. One of the means allowing meta-modeling platforms to communicate is based on open standards. Products implementing the common protocols defined in the standard are thus interoperable by design. By providing users with a freedom to start their implementation of a modeling method on one platform, continue it on the second, and finish it on the third is a tangible benefit.

Scalability [42] allows handling a growing amount of work in a capable manner. This issue can be illustrated on an example where one provides an implemented modeling method as a service that needs to scale up with the number of users.

E. Analyzing Best Practices and Guidelines for the Design of DSLs

Designers need to avoid the common undesirable features that make the learning and using of the language harder than it should be. A substantial part of this difficulty arises from the structure, syntax and semantics of a language. Desirable and undesirable features also raise requirements for the very nature of a DSL.

Programming language designers are programming experts typically far removed both temporally and cognitively from the difficulties experienced by novice programmers. This can result in languages that are either too restrictive or too powerful (or sometimes, paradoxically, both) [43]. To avoid falling into a trap of designing a language only highly trained experts are able to use efficiently, we have collected a couple of key desirable language features, as well as a couple of undesireable ones, which should be avoided if possible. According to [43] the most notable desirable features are:

- *User expectation conformity*. Languages should be designed so that reasonable assumptions based on prior non-programing-based knowledge (e.g., domain expert knowledge) remain reasonable assumptions in the programming domain, meaning that the constructs of a language should not violate user expectations;
- *Readable and consistent syntax*. By choosing the constructs with which the recipient is already familiar (e.g. 'if' rather than 'cond', 'head/tail' rather than 'car/cdr') syntactic noise can be minimized; on one hand, reducing syntactic noise might involve minimizing the overall syntax; alternatively, it may be better to increase the complexity of the syntax in order to reduce homonyms which blur the signal;
- *Small and orthogonal set of features*. A small non-overlapping set of language features with distinct and mnemonic syntactic representations and with semantics which mirror as closely as possible the real-world concepts; features that are not necessary should not be included in the language;
- *Error diagnosis*. Without good error detection and debugging support users can spent hours trying to decipher why isn't the program doing what it is intended; on the other hand error messages should be meaningful and without unnecessary technical jargon.

It also helps if the designer is knowledgeable in the domain for which the language is being developed, thus having better awareness of the real-world concepts that need to be included into the language and the ways these concepts are expressed.

Some of the most undesirable features from the language user's perspective, according to the same source are:

- *Paradigmatic purity*. Strict adherence to a single functional, logical or object oriented paradigm can make for a certain conceptual simplicity and elegance, but in practice it can also lead to extremely obscure and unreadable code; in some cases relatively simple programs must be substantially restructured to achieve even basic effects;
- *Language bloat***.** Extreme complexity and a large palette of features might seem as a good idea at first, but they come together with a steeper learning curve, higher level of confusion, difficulties of adequate error detection, very complex syntax and semantics;
- *Syntactic synonyms*. Two or more syntaxes are available to specify a single construct; common example is dynamic array access in C, where the second element of an array may be accessed by any of the following syntaxes, some of which are legal only in certain contexts: array[1], *(array+1), 1[array], *++array;
- *Syntactic homonyms*. Constructs which are syntactically the same, but have two or more different semantics depending on the context are perhaps a more serious flaw in a language then syntactic synonyms; an extreme example may be seen in Turing, in which the construct A(B) has five distinct meanings, but not as extreme as LISP and its variants, which can be viewed as one massive homonym;
- *Hardware dependency*. There seems to be no convincing reason why the user, already struggling to master syntax and semantics of various constructs, should also be forced to deal with details of representational precision, varying machine word sizes, or awkward memory models; the data types are particularly problematic in C as they are generally not portable, for example, the standard int type varies from 16-bit to 32-bit representations depending on the machine and the implementation; this can lead to strange and unexpected errors when overflow occurs;

- *Backward compatibility*. This property is surely useful from the experienced programmer's point of view, as it promotes reuse of both code and programming skills, but one needs to be careful, because it constraints the design of a new language; Stroustrup [44] acknowledged this problem: *"Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility."*

In the long run some of these features are very hard to avoid, especially if there was no plan for language evolution in the design phase. Taking a systematic approach and considering future needs of language users plays a significant role in the further development of a language.

F. The Evolution Requirement

Languages evolve and this is also relevant for DSLs [45]. Evolution, in this context, does not only mean that one language has changed over a period of time. It also means that new languages have been created using some of the concepts from older, already existing, languages. Good examples are C++ and C# evolving from C, JavaScript starting from concepts used in Java, etc.

Nowadays, a considerable amount of new languages are designed by a single person or a small team, making DSLs a popular paradigm with other reasons being: affordable investments, standardization is not a necessity as it used to be in the past, out-of-the-box tools for development of tailored languages, the Internet as a medium for distribution and user feedback and requests. Evolution can be enabled if the DSL has been designed to follow the progress of technology it depends upon, and predict the future changes in the domain it describes.

Concerning the hereby proposed DSL, one should consider the following: (1) the future development of meta-modeling platforms, and (2) the possible changes in the application domain, which can come from various sources: new findings in the academia or industry, insights during the use of the DSL. To be able to cope with the upcoming issues, the DSL under consideration should have at least these important features: (1) extensible abstract syntax, concrete syntax and semantics, and (2) extensible execution engine.

Extensible abstract syntax allows modification, removal or addition of concepts. The same is true for extensible concrete syntax, and semantics. It is also important to have a mapping mechanism between abstract and concrete syntax, and between abstract syntax and semantics, which should support extensions as well.

The execution engine (compiler) is responsible for transforming code written in the DSL to the format that can be run on a meta-modeling platform. Therefore, it is essential that changes done to the target platforms can be implemented into the engine.

IV. The Grammar Of the Domain-Specific Modeling Method Definition Language

A first draft of the introduced DSL's context-free grammar in formal EBNF form was made available at [46], with the current section providing only an overview of it.

The language is designed around the concepts of *inheritance* (reuse of characteristics from parent classes, typically of abstract nature, to child classes, to be instantiated in models) and *referencing* (reusing previously defined objects by passing their identifiers to other constructs).

Table 2 provides an excerpt of the language grammar, highlighting the main statements of the language. A design decision was made to avoid the tagging overhead of an XML-based syntax and to opt for a simple declarative style that can accommodate easily existing code editing approaches (XText [20]) and algorithm description approaches (XBase [47]).

TABLE II. Excerpt from the EBNF Grammar of a DSL for Modeling Method Definition

Statement	Statement Specification in EBNF	Meaning
Root	root ::= methodname embedcode* method	The root of the method definition document
Method Name	methodname ::= 'method' name	
Embed	embedcode ::= 'embed' name '<' name-embedplatformtype (':' name-embedcodetype)? '>' 'start' embeddedcodegoeshere 'end'	In the case that native code for the target metamodeling platform will be embedded, the platform must be declared.
Method	method ::= enumeration* symbolstyle* symbolclass* symbolrelation* metamodel algorithm* event*	Container for the method building blocks and auxiliary elements
Enumeration	enumeration ::= 'enum' name '{' enumvalues+ '}'	An auxiliary element of a method definition, defining a list of values (typically used to restrict attributes of modeling objects)
Metamodel	metamodel ::= class+ relation* attribute* modeltype+	The main building block of a method, describing structurally the language metamodel
Class	class ::= 'class' name ('extends' name-class)? ('symbol' symbolclass)? '{' (attribute | insertembedcode)* '}'	The definition of a modeling concept, including assignment of its graphical notation (if instantiable), its editable property set and prescribed inheritance, to be instantiated by modeling objects
Relation	relation ::= 'relation' name ('extends' name-relation)? ('symbol' name-symbolrelation)? 'from' name-class 'to' name-class '{' (attribute | insertembedcode)* '}'	The definition of a modeling relation, including assignment of its graphical notation, its editable property set and prescribed inheritance
Attribute	attribute ::= 'attribute' name ':'	The definition of a

	type ('access' ':' acesstype)?	property (for a modeling concept or relation)
Access	acesstype ::= 'write' | 'read' | 'internal'	The definition of the access mode for a property
Model Type	modeltype ::= 'modeltype' name '{' 'classes' name-class+ 'relations' ('none' | name-relation+) 'modes' ('none' | name-mode+) '}'	The definition of a model type as a partition of the language metamodel
Mode	mode ::= 'mode' name 'include' 'classes' name-class+ 'relations' ('none' | name-relation+)	The definition of a mode (view) on a model type limiting its available constructs
Class Symbol	symbolclass ::= 'classgraph' name ('style' name-symbolstyle)? '{' (svgcommand | insertembedcode)* '}'	The definition of graphical notations and styles for modeling concepts
Relation Symbol	symbolrelation ::= 'relationgraph' name ('style' name-symbolstyle)? '{' 'from' (svgcommand | insertembedcode)* 'middle' (svgcommand | insertembedcode)* 'to' (svgcommand | insertembedcode)* '}'	The definition of graphical notations and styles for modeling relations
SVG Command	svgcommand ::= (rectangle | circle | ellipse | line | polyline | poligon | path | text) symbolstyle	The SVG-style description of notation elements (language symbols)
Symbol Style	symbolstyle ::= 'style' name '{' 'fill' ':' ('none' | fillcolor) 'stroke' ':' strokecolor 'stroke-width' ':' strokewidth ('font-family' ':' fontfamily)? ('font-size' ':' fontsize)? '}'	The definition of graphical styles to be applied on notations
Algorithm	algorithm ::= 'algorithm' name '{' (algorithmoperation | insertembedcode)* '}'	The description of a modeling algorithm (including the possibility of embedding native code from the target platform)
Event	event ::= 'event' name-event '.' 'execute' '.' name-algorithm	The definition of an algorithm trigger

V. PROOF-OF-CONCEPT

The hereby introduced concept has been evaluated with respect to feasibility by implementing it in a modeling method definition environment (based on XText) together with a compiler for the ADOxx meta-modeling platform. The editor includes compile time error checking, code autocompletion, highlighting, and reusable code templates. Tests have been performed with (additively) evolving requirements. A minimal representative example will be showcased in this section, reflecting the final set of requirements for a "car parking" modeling method to be detailed below.

The assumed modeling requirements are that a tool should allow the modeler to describe (a) courier tasks as sequences of actions and path-splitting decisions (hence they can be considered rudimentary workflows); (b) the allocation of parking spaces of different types to geographical areas (cities); (c) mappings between the various steps of a courier task to the geographical areas where they must be performed. Actually these can be considered already early design hints – the originating requirements are rather query-oriented (e.g. *I want to be able to retrieve the list of all parking areas required after a particular courier decision*) or functional-oriented (e.g. *I want to be able to generate a list of parking objects from a list of parking identifiers*).

Fig. 5 grounds this in design decisions at metamodel level, also capturing some key representative features: the language is partitioned into two model types – one for courier tasks, one for city-parking area-parking type allocations. Semantics are expressed (a) as property sets for both objects (e.g. a city has a country) and relations (the condition of the *Next* "arrow"); (b) as class specializations based on some abstract classes (e.g. everything in a courier task is a Node, which has a name and a universal identifier); (c) as navigable relations between models of different types (the *requiresParkingInCity* hyperlink between courier actions and cities). Not reflected in the metamodel, there is also the requirement of generating modeling objects from a list of universal identifiers (the URI attribute inherited by all concepts from the abstract RootClass).

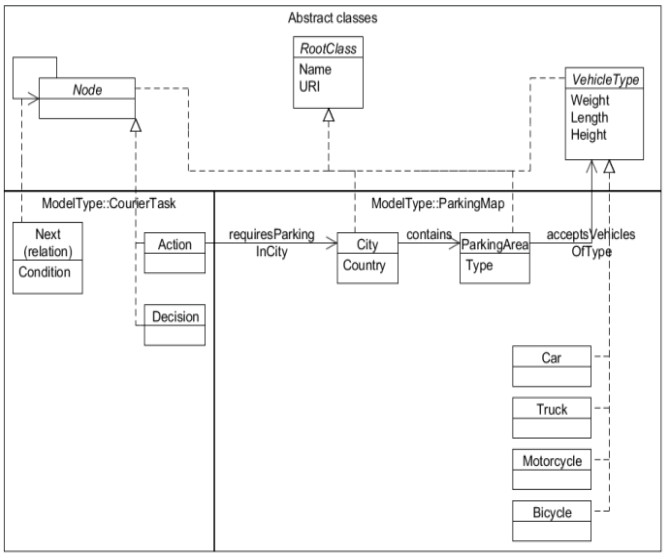

Fig. 5. Metamodel of the running example

The final outcome is the modeling tool visible in Fig. 6, where one sample of each model type is shown (*OnDemandTask* of type *CourierTask*, *Parkings* of type *ParkingMap*), together with screenshots of property sets for various elements (e.g. the *Condition* of the *Next* arrow, the *Country* of the *City*) and the cross-model hyperlink between *Actions* and *Cities* (*requiresParkingInCity*).

The code following the figure illustrates the building blocks of this modeling method definition (code comments highlight some key features). It is important to note that the compiler generates (as much as possible) defaults wherever the method definition is lacking mandatory elements. This is particularly relevant in graphical notations – wherever they are omitted, shapes of random size and color will be generated, which is a nice feature for fast prototyping focused on the language semantics rather than visualization. It is also a fallback mechanism for meta-modeling platforms that might not support

features provided by the DSL (e.g. a platform that does not support dynamic vectorial notation definition, only static bitmaps).

Also, a particular approach to extensibility and interoperability is noticeable in this example: the possibility of *embedding native code* of the targeted meta-modeling platform (ADOxx in this case) for features that are not (temporarily or by design) supported by the DSLs compiler. This is applied here for the algorithm that generates modeling objects from imported identifiers, which falls back on native ADOxx code (for the general case, the platform-independent algorithm description syntax is based on XBase [47]).

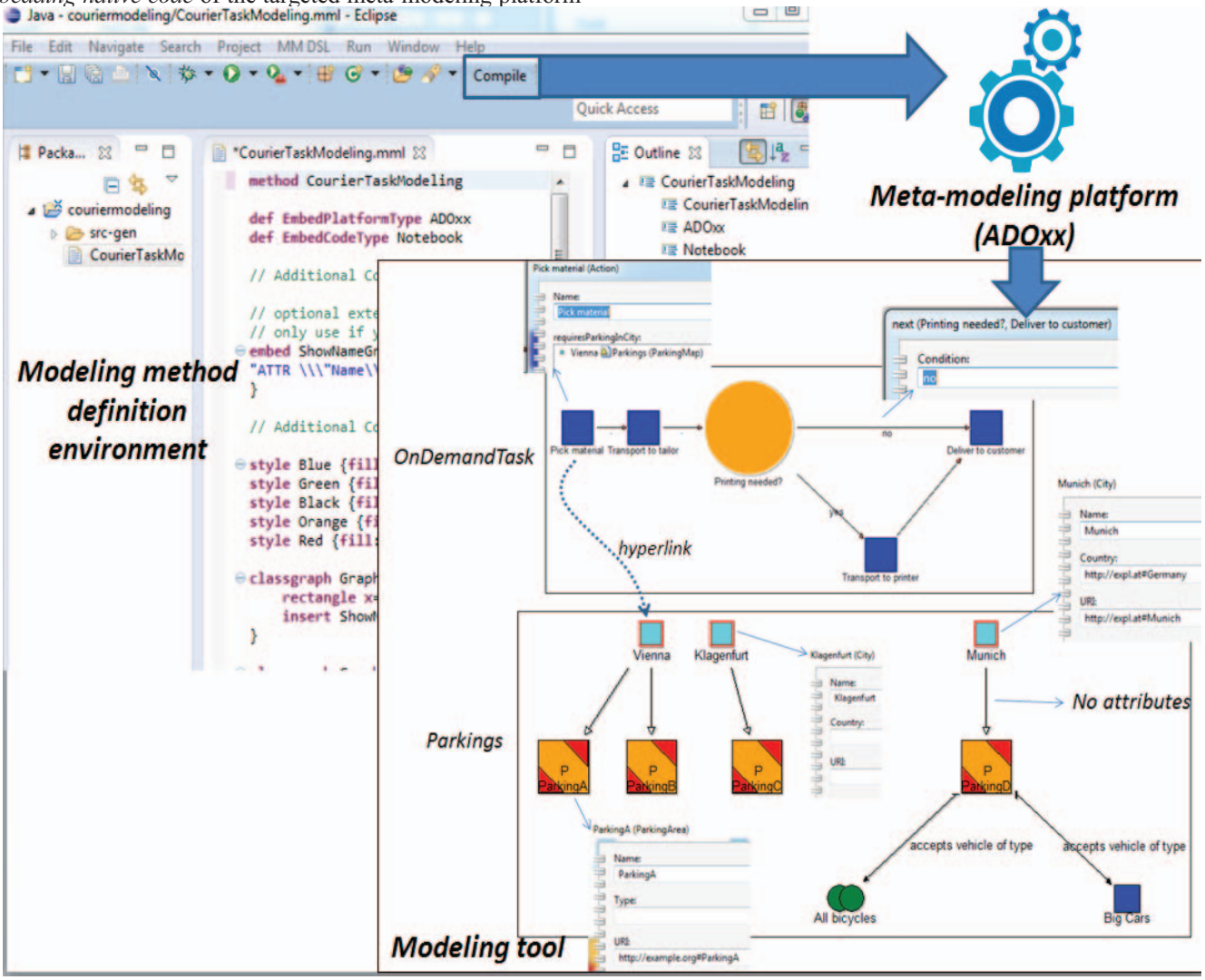

Fig. 6. Model samples created with the modeling method implementation of the running example

```
method CourierTaskModeling

def EmbedPlatformType ADOxx
embed ShowNameGraph <ADOxx:Notebook>
 {"ATTR \\\"Name\\\" x:0pt y:9pt w:c"}
embed URIImportAlgorithm <ADOxx:AdoScript>
{ "
SETL cls_name:(\\\"ParkingArea\\\")
SETL mod_type_name:(\\\"ParkingMap\\\")
SETL attr_uri_name:(\\\"URI\\\")
SETL obj_cnt:(0)
// native ADOxx code is embedded to compensate missing features

….}

style Orange {fill:orange stroke:black stroke-width:1}
style Red {fill:red stroke:black stroke-width:1}
…
```

```
// style definitions to be used in notations

classgraph GraphParkingArea {
      rectangle x=-20 y=-20 w=40 h=40 style Orange
      polygon points=-20,20 0,20 -20,0 style Red
      polygon points=0,-20 20,-20 20,0 style Red
      text "P" x=-2 y=-2
      insert ShowNameGraph
}
…
// notation definition for one modeling class (if this section is omitted, shapes of random size and color are generated by the compiler for each class

relationgraph GraphNext style Red{
      from
      middle
      insert DynConditionGraph
```

```
        to
        polygon points=-2,2 2,0 -2,-2
    }
    // notation definition for one modeling relation

    class Root
        {attribute URI:string}
    // abstract class definition (name attributes are generated by default, so they don't have to be explicitly defined)

    class ParkingArea extends Root symbol GraphParkingArea
        {attribute Type:string}
    // property inheritance (from Root) and assignment of graphical notation to a modeling class

    class Action extends Node
        {reference requiresParkingInCity -> modeltype ParkingMap class City}
    // definition of hyperlink between models of different types

    relation acceptsVehiclesOfType symbol GraphAcceptsVehicles
    from ParkingArea to VehicleType {}
    // definition of visual modeling relation

    relation next symbol GraphNext from Node to Node
        {attribute Condition:string}

    modeltype ParkingMap {
        classes City ParkingArea Car Truck Motorcycle Bicycle
        relations acceptsVehiclesOfType contains
        modes none}
    // grouping language constructs into a model type

    algorithm URIImport {
        insert URIImportAlgorithm}
    // embedding native ADOxx functionality as algorithms, due to the temporary unavailability of the algorithm description component of the grammar
```

VI. Conclusive SWOT Evaluation

After the research conducted on several different areas - modeling methods, meta-modeling platforms and various computer language design principles - a wide range of requirements have been collected as the starting point for designing a DSL for the application domain of *modeling method engineering*, with the main motivation being (a) to provide an ability of quickly editing a modeling method that must evolve agilely and (b) to do it in a platform-independent manner, which can be mapped through platform-specific compilers to any targeted meta-modeling environment.

Further on, a language grammar has been formulated for the envisioned DSL in terms of EBNF, validated through an XText-based implementation of a programming environment which includes a code editor and a compiler for the ADOxx meta-modeling platform.

Based on this proof-of-concept implementation, a SWOT evaluation has been performed:

Strengths: The proposal treats modeling method engineering as an application domain for a domain-specific language that enables code-based platform-independent definition of conceptual modeling methods. The language was designed flexible enough to include native code for the targeted meta-modeling platform if its features prove insufficient for specific cases (at least until a new version is developed);

Weaknesses: Compilers need to be written in order to deploy the modeling method definition on popular meta-modeling platforms and produce the final outcome – modeling tools. Currently only a compiler for ADOxx was realized, employed in the presented proof-of-concept. Other compilers may emerge in the future from individual or community-based efforts invited and fostered through the research environment of the Open Model Initiative Laboratory, which is a medium-term goal;

Opportunities: A "write-once, run-everywhere" principle, as well as improved portability, may be embraced for modeling method engineering if compilers emerge for popular meta-modeling platforms. This may have significant impact on productivity and fast prototyping, as well as on putting forth agility as a key quality attribute for modeling methods;

Threats: The proposed DSL adds a new abstraction layer to meta-modeling, one that requires its own learning curve. Methodologists familiar with a particular meta-modeling platform may prefer to specialize and support further evolution of the preferred platform rather than taking the necessary distance to approach modeling method engineering through platform-independent programming. Again, availability of additional compilers and case-based demonstrations or requirements are key factors in advocating the versatility of the hereby proposed approach.

The **takeaway message** is that quickly evolving modeling requirements need enablers for fast prototyping across metamodeling platforms. The concept of *modeling method* should be platform-independent in the sense that methodologists should work primarily with concepts representing its building blocks, then use existing platforms to deploy and fine tune then, rather than having the modeling method as a byproduct of an implementation effort for a specific platform. This view highlights the importance of pre-implementation phases in modeling method engineering (relative to Fig. 1) - just like in general software engineering, platform-independent design decisions should propagate to platform-specific implementation and deployment, and not the other way around.

References

[1] D. Karagiannis and H. Kühn, "Metamodelling Platforms," in E-Commerce and Web Technologies, vol. 2455, K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 182.

[2] H.-G. Fill and D. Karagiannis, "On the Conceptualisation of Modelling Methods Using the ADOxx Meta Modelling Platform", Enterprise Modelling and Information Systems Architectures, Vol. 8, Issue 1, 2013, pp. 4-25.

[3] "OMG's MetaObject Facility (MOF) Home Page." [Online]. Available: http://www.omg.org/mof/. [Accessed: 8-Feb-2015].

[4] "Open Model Initiative Laboratory" [Online]. Available: http://www.omilab.org/. [Accessed: 8-Feb-2015].

[5] "Object Management Group - UML." [Online]. Available: http://www.uml.org/. [Accessed: 8-Feb-2015].

[6] "BPMN Information Home." [Online]. Available: http://www.bpmn.org/. [Accessed: 8-Feb-2015].

[7] F.B. Aydemir, P. Giorgini, J. Mylopoulos, F. Dalpiaz, "Exploring Alternative Designs for Sociotechnical Systems", in M. Bajec, M.

Collard, R. Deneckere (eds.), Proceedings of the 8th Int. Conf. RCIS, 2014, IEEE, pp. 1-12

[8] M. Ruiz, S. Espana, O. Pastor, "Supporting Organisational Evolution by Means of Model-Driven Reengineering Frameworks", in R. Wieringa, S. Nurcan, C. Rolland, J.L. Cavarero (eds.), Proceedings of the 7th IEEE Int. Conf. RCIS, 2013, IEEE, pp. 1-10

[9] H.-G. Fill, T. Redmond, D. Karagiannis, "FDMM: A Formalism for Describing ADOxx Meta Models and Models", in: L. Maciaszek, A. Cuzzocrea and J. Cordeiro: Proceedings of ICEIS 2012 - 14th International Conference on Enterprise Information Systems, Vol.3, SciTePress, 2011, pp. 133-144.

[10] "The ADOxx meta-modeling platform" [Online]. Available: http://www.adoxx.org/. [Accessed: 8-Feb-2015].

[11] C. Atkinson, M. Gutheil, and B. Kennel, "A Flexible Infrastructure for Multilevel Language Engineering," IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 742–755, Dec. 2009.

[12] M. P. Ward, "Language Oriented Programming," SOFTWARE—CONCEPTS AND TOOLS, vol. 15, pp. 147–161, 1995.

[13] T. Clark, P. Sammut, and J. Willans, "Applied metamodelling: a foundation for language driven development.," 2008. [Online]. Available: http://eprints.mdx.ac.uk/6060/. [Accessed: 23-Sep-2011].

[14] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003.

[15] D. Ameller, X. Franch, and J. Cabot, "Dealing with Non-Functional Requirements in Model-Driven Development," in Requirements Engineering Conference (RE), 2010 18th IEEE International, 2010, pp. 189 –198.

[16] J. Greenfield and K. Short, Software factories: assembling applications with patterns, models, frameworks, and tools. Wiley Pub., 2004.

[17] T. Clark, P. Sammut, and J. Willans, "Superlanguages: developing languages and applications with XMF.," 2008. [Online]. Available: http://itcentre.tvu.ac.uk/~clark/docs/Superlanguages.pdf. [Accessed: 15-Jan-2013].

[18] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html. [Accessed: 8-Feb-2015].

[19] M. Fowler and R. Parsons, Domain Specific Languages, 1st ed. Addison-Wesley Longman, Amsterdam, 2010.

[20] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, New York, NY, USA, 2010, pp. 307–309.

[21] "Irony - .NET Language Implementation Kit. - Home." [Online]. Available: http://irony.codeplex.com/. [8-Feb-2015].

[22] "Microsoft Visual Studio 2010 Visualization & Modeling SDK," Microsoft Download Center. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=23025. [Accessed: 8-Feb-2015].

[23] J.-P. Tolvanen and S. Kelly, "MetaEdit+: defining and using integrated domain-specific modeling languages," in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, New York, NY, USA, 2009, pp. 819–820.

[24] D. Karagiannis and N. Visic, "Next Generation of Modelling Platforms," in Perspectives in Business Informatics Research, vol. 90, J. Grabis and M. Kirikova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–28.

[25] "Graphiti". [Online]. Available: http://www.eclipse.org/graphiti/. [Accessed: 8-Feb-2015].

[26] T. Clark: Xmodeler Homepage. [Online]. Available: http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/XModeler.html . [Accessed: 8-Feb-2015].

[27] "Petri Nets." [Online]. Available: http://www.petrinets.info/. [Accessed: 8-Feb-2015].

[28] T. Koshy, Discrete Mathematics with Applications. Academic Press, 2004.

[29] H. Kern, A. Hummel, and S. Kühne, "Towards a comparative analysis of meta-metamodels," in Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11; VMIL'11, New York, NY, USA, 2011, pp. 7–12.

[30] "The OWL 2 Language Primer" [Online]. Available: http://www.w3.org/TR/owl2-primer/. [Accessed: 8-Feb-2015].

[31] "Open Model Initiative Projects" [Online]. Available: http://www.omilab.org/web/guest/projects. [Accessed: 8-Feb-2015].

[32] Y. Moschovakis, What Is an Algorithm? 2001.

[33] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, Introduction to Algorithms, 2nd ed. McGraw-Hill Higher Education, 2001.

[34] "The CIDOC CRM." [Online]. Available: http://www.cidoc-crm.org/. [Accessed: 8-Feb-2015].

[35] D. L. Moody, P. Heymans, and R. Matulevicius, "Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of i* Visual Syntax," in Requirements Engineering Conference, 2009. RE '09. 17th IEEE International, 2009, pp. 171 – 180.

[36] S. Islam, H. Mouratidis, and J. Jürjens, "A framework to support alignment of secure software engineering with legal regulations," Softw. Syst. Model., vol. 10, no. 3, pp. 369–394, Jul. 2011.

[37] H.-G. Fill, "On the Conceptualization of a Modeling Language for Semantic Model Annotations," in Advanced Information Systems Engineering Workshops, vol. 83, C. Salinesi and O. Pastor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 134–148.

[38] M. Glinz, "Very Lightweight Requirements Modeling," in Requirements Engineering Conference (RE), 2010 18th IEEE International, 2010, pp. 385–386.

[39] "OMILab auxiliary tools and services" [Online]. Available: http://www.omilab.org/web/guest/tools. [Accessed: 8-Feb-2015].

[40] M. Völter and E. Visser, "Language extension and composition with language workbenches," in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, New York, NY, USA, 2010, pp. 301–304.

[41] H. Kühn and M. Murzek, "Interoperability Issues in Metamodelling Platforms," in Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications, Geneva, 2006, pp. 215–226.

[42] K. Sledziewski, B. Bordbar, and R. Anane, "A DSL-Based Approach to Software Development and Deployment on Cloud," in 2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2010, pp. 414–421.

[43] L. McIver and D. Conway, "Seven deadly sins of introductory programming language design," in Software Engineering: Education and Practice, 1996. Proceedings. International Conference, 1996, pp. 309–316.

[44] B. Stroustrup, The design and evolution of C++. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.

[45] "Research in Programming Languages | Tagide." [Online]. Available: http://tagide.com/blog/2012/03/research-in-programming-languages/. [Accessed: 8-Feb-2015].

[46] "The MM-DSL grammar specification" [Online]. Available: http://www.omilab.org/c/document_library/get_file?uuid=eb040aac-ea0d-4df7-a0a9-80b73f00c5f8&groupId=10122, [Accessed: 8-Feb-2015].

[47] "Generic XBase grammar for algorithm descriptions" [Online]. Available:https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtext.xbase/src/org/eclipse/xtext/xbase/Xbase.xtext. [Accessed: 8-Feb-2015].