UDC 004.658.6, DOI:10.2298/CSIS090203002D

A Domain-Specific Language for Defining Static Structure of Database Applications

Igor Dejanović, Gordana Milosavljević, Branko Perišić, and Maja Tumbas

Faculty of Technical Sciences, Trg D. Obradovića 6,
21000 Novi Sad, Serbia
{igord,grist,perisic,majat}@uns.ac.rs

Abstract. In this paper we present DOMMLite - an extensible domain-specific language (DSL) for static structure definition of database-oriented applications. The model-driven engineering (MDE) approach, an emerging software development paradigm, has been used. The language structure is defined by the means of a metamodel supplemented by validation rules based on Check language and extensions based on Extend language, which are parts of the openArchitectureWare framework [1]. The metamodel has been defined along with the textual syntax, which enables creation, update and persistence of DOMMLite models using a common text editor. DSL execution semantics has been defined by the specification and implementation of the source code generator for a target platform with an already defined execution semantics. In order to enable model editing, a textual Eclipse editor has also been developed. DSL, defined in this way, has the capability of generating complete source code for GUI forms with CRUDS (Create-Read-Update-Delete-Search) and navigation operations [2,3,4,5].

Keywords: DSL; Domain-specific; MDE; MDSD; MDA; CRUD; Modeling; Meta-modeling; Generator.

1. Introduction

One of the issues that continues to pose difficulties for computer engineers and developers is increasing complexity of software and supporting hardware architecture. A variety of different methods has been employed in an attempt to overcome these issues, but what they all have in common is raising the level of abstraction. Although powerful, used abstraction are usually computer-, i.e. solution space-oriented, as opposed to being application domain-, i.e. problem space-oriented [6]. Developers still need to perform the mental mapping of concepts found in the solution domain to concepts found in the problem domain and to apply these mappings manually during the course of implementation [7].

The use of computer related concepts in solving real-world problems leads to all sorts of problems:

- Communication issues arise between technology and business domain experts. They usually have different interpretations of concept semantics, which leads to errors in early phases of software development.
- Mapping between the application domain and the technology domain is an error-prone manual process.
- A minor change in business domain requirements can lead to huge changes in the technology domain layer.
- Rapid pace of technology changes renders applications out-of-date and leads to constant need for migration to new platforms, or new versions of the same platform, which increases maintenance costs.

The importance of using domain-specific concepts in software development is explained in [8]:

"Perhaps the greatest difficulty associated with software development is the enormous semantic gap that exists between domain-specific concepts encountered in modern software applications, such as business process management or telephone call processing, and standard programming technologies used to implement them.

...

Clearly, the more directly we can represent concepts in the application domain, the easier it becomes to specify our systems.
Conversely, the greater the distance between the application domain and the model, the less value we get from modeling."

In this paper we present DOMMLite, a domain-specific language (DSL) for static structure definition of database-oriented applications. The purpose of this paper is to give an overview of the DOMMLite language and its supporting tools and to address some issues and choices that have been made in the design of the language. It is by no means a full specification of the language; therefore, we provide some insight into the differences between DOMMLite and other OO modeling languages and consider common concepts known from other languages. A full specification of the DOMMLite language is given in [9].

The language has been designed and implemented using model-driven engineering (MDE) techniques, which are a specialization of DSL engineering techniques [10]. DOMMLite builds on the concepts of other object-oriented modeling languages such as UML [11], MOF [12], ECore [13], and concepts expressed in Domain-Driven Design [14]. It is a declarative language and, although many of its constructs more or less resemble those found in other OO modeling languages, there are differences that will be identified in the rest of the paper; therefore, DOMMLite is not created by mere extension or restriction of existing languages/meta-models. Having that in mind we state

that, according to the classification introduced by M. Fowler (see Sect. 2), DOMMLite is an external language.

The purpose of the language is to enable developers to specify, in a simple manner, static structure of database oriented applications with enough meta-data to generate fully working applications with implementation of CRUDS (Create-Read-Update-Delete-Search) operations without resorting to the more heavy-weight modeling languages such as UML. Both the abstract syntax, defined by a metamodel, and one of the possible concrete syntaxes have been developed. The implementation of the abstract and the concrete textual syntax of the language, the model editor and the source code generator has been carried out using the openArchitectureWare (oAW) generator framework [1], a metamodel agnostic framework which is well integrated with the Eclipse Modeling Framework [13]. For the purpose of this work a textual concrete syntax was created. Although creating a graphical syntax and a graphical editor is made a lot easier with projects such as GMF[1] and GEMS[2], the creation of a fully featured graphical editor in the context of changing requirements still requires a considerable amount of time. This is why work on further improvements of the textual syntax will continue until the language is stable enough. DOMMLite defines the execution semantics by the implementation of the source code generator for a target platform with an already defined execution semantics.

The rest of the paper is structured as follows. Section 2 gives overview of DSLs as the underpinning technique used in this paper. In section 3 the abstract syntax of the DOMMLite language is described. Section 4 gives an overview of the language's concrete syntax based on the xText [15] language. Section 5 explains the design and implementation process of the application code generator. Section 6 points out several issues regarding the completion of the eclipse editor generated by the oAW framework. Section 7 analyzes related work. Section 8 gives final conclusions.

2. Domain-Specific Languages

Domain-specific languages, in contrast to general-purpose languages (GPL), offer, through specific notations and abstractions, the power of expression focused on, and usually restricted to, a particular problem domain [16].
Some of the advantages of using DSLs over GPLs are:

- DSLs are usually more concise and expressive than GPLs, which enables programmers to represent their intentions more clearly.
- DSL syntax, both textual and graphical, can be tailored to the specific knowledge of the domain experts.

1 http://www.eclipse.org/gmf/
2 http://www.eclipse.org/gmt/gems/

- Concepts used in DSLs are found in the application domain, so their use does not require domain experts to have programming skills.
- Expressing the domain construct through concepts independent of the technology used results in a longer lifespan of the application. If applied correctly, application description needs to change only if business requirements change and is immune to changes in the technology layer, which can be handled by the application generator (DSL compiler).
- Using higher- level abstraction leads to the reduced number of lines of code (LOC) (in terms of textual syntaxes), which has a positive impact on the development and maintenance. Some researchers achieved a 50:1 ratio of LOC in favor of DSLs [17]. Software fault density (number of software faults per one thousand lines of code) does not significantly depend on the language being used [18]. Therefore, using DSL languages reduces the number of software bugs, which leads to increased software quality and lower maintenance costs.
- In situations where code analysis, verification, optimization, parallelization, and transformation techniques are considered to be very difficult or almost impossible to achieve with GPLs, it is possible to achieve them in the context of DSLs [19].

Better expressiveness of DSL languages does not come for free. For the sake of it they give away their generality [19], so DSLs are usually not very useful outside of the domain they were constructed for.

Some programming languages started out as DSLs, but have evolved towards GPLs by getting more features. A reverse process has not been observed in the history of programming languages [10].

There are also widespread languages that are essentially DSLs, although they may not be known as such. Examples of such languages include HTML - the language for describing hypertext documents, which forms a foundation of today's global network, SQL - a structured language for querying, updating and deleting data in relational databases, LaTeX - a language for document typesetting. Even Spring's[3] XML-based configuration file language can be considered a DSL for expressing application configuration.

Different classifications of DSLs exist. DSLs can be classified in the same way as GPLs. For example, they can be classified as: object-oriented or functional, imperative or declarative, visual or textual.

Another classification, by the way DSLs are constructed, is given by Martin Fowler in [20].

He divides them into two groups:

3 http://www.springframework.org/

- External DSLs - These languages are also called little languages and are very popular in the Unix community. Representatives include awk, sed, flex, yacc etc. Their main property is that they are built from scratch, with their syntax carefully tailored for the domain in question.
- Internal DSLs - In contrast to external DSLs, internal DSLs are built on top of an existing GPL, extending their syntax to add support for domain-specific constructs. They are gaining popularity with the emergence of GPLs which have mechanisms for easy extensibility. Some of the representatives are Ruby[4] [21], Scala[5], or Python[6] [22]. There are also internal DSLs based on main-stream programming languages like Java [23]. Using this approach one can get an entire tool-chain of the host language for free (editors, compilers, debuggers etc.). Internal DSLs are also called domain-specific embedded languages [24], or simply embedded languages [16].

Fowler's classification can also be applied in the context of modeling languages. An external DSL in the context of modeling technologies and visual syntaxes is usually referred to as domain-specific modeling language (DSML) [7]. Similarly, a UML profile-based modeling language is, according to Fowler's classification, an internal DSL.

A completely clear distinction between DSLs and GPLs does not exist, although attempts have been made to construct a method for quantifying domain specificity [25]. As stated in [19], domain specificity is a matter of degree: it largely depends upon the notion of a domain.

For the purpose of this paper we will use the definition of the DSL based on MDE ideas, given in [10], as follows.

A DSL is a set of coordinated models:

- Domain definition metamodel(DDMM) - is a conceptualization of the domain that introduces the basic abstractions of the domain and their mutual relations. Once such an abstract entity is explicitly represented as a model, it becomes the reference model for the models expressed in the DSL, that is, it is a metamodel. That metamodel is referred to as the domain definition metamodel (DDMM).
- Concrete syntax - represents a transformation of DDMM to the "display surface" metamodel. More than one such transformation can be defined, or to put it simply, for each DDMM multiple concrete syntaxes can be defined, both textual and visual.
- Semantics - A DSL can have execution semantics defined. Semantics is also defined by a transformation from DDMM into a

4 http://www.ruby-lang.org/
5 http://www.scala-lang.org/
6 http://www.python.org

DSL or GPL which has an already defined precise execution semantics.

Using this definition as the basis, the following section defines DDMM, the concrete syntax and semantics of DOMMLite.

3. Abstract Syntax

The abstract syntax of the DOMMLite language is defined by a metamodel. A great deal of DOMMLite is inspired by other OO based metamodels, such as MOF, UML and ECore, and in many ways it builds on top of the concepts from these reputable languages. The concrete implementation of DOMMLite is based on the Eclipse Modeling Framework (EMF[7]), an OO (meta)modeling infrastructure. The abstract syntax of the language will be presented using UML class diagram notation. The most important conceptual primitives of the DOMMLite language are based on well-known concepts described in [14].

The semantics of concepts in DOMMLite is given in the form of recommendations. The model compiler defines the semantics in detail, so the language's full semantics interpretation is left to the compiler developer. Recommendations for semantics interpretation described in the following sections have been followed in the implementation of the DOMMLite model compiler prototype (see Sect. 5). This section gives an overview of some of the most important conceptual primitives of the DOMMLite language.

3.1. Data Types

The *DataType* metaclass (see Fig. 1) defines, in DOMMLite terminology, simple types of the DOMMLite language. Simple types have no internal structure, which distinguishes them from complex types (e.g. entities, services etc) that do. *UserDataType* and *BuiltInDataType* are also referred to as primitive types. Primitive types can be built-in (defined by the language itself) or user-defined. Built-in types are: *void, bool, int, real, money, string, char, date, datetime*. User-defined types can be defined by the modeler and used troughout the model in the same way as the built-in ones. The semantics of the types is implemented on the target platform. The semantics of built-in types is implemented once for the given platform and can be reused in many projects without change. Following definitions in [26], it is a part of "the platform". The user-defined type semantics is specified on the target platform for the project where it is introduced in the model. If the source code generator is carefully tailored, it is usually not necessary to make

7 http://www.eclipse.org/modeling/emf/

changes to the generator itself in order to support the newly defined type (see Sect. 5 for an example of the generator prototype).

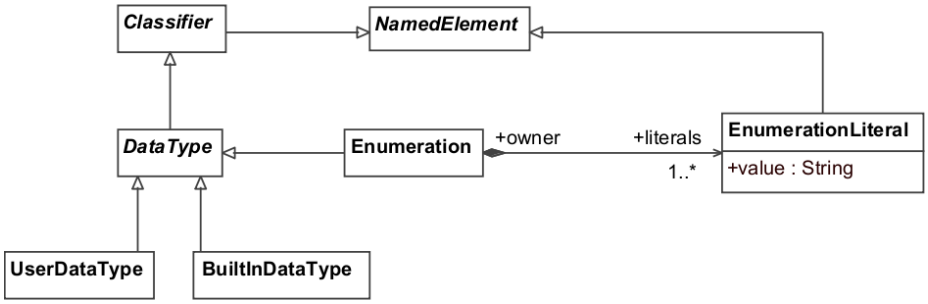

Fig. 1. The DataType metaclass

3.2. Model Organization

The DOMMLite language is organized in a structural manner using the notion of packages (see Fig. 2). The *Package* metaclass in DOMMLite has the semantics similar to that of the *Package* metaclass in UML, but it is implemented differently. The language elements that are organized in this way must inherit from the *PackageElement* metaclass. In order to support package nesting, the *Package* metaclass also inherits from the *PackageElement*. The model itself is described by the *DOMMLiteModel* metaclass. Instances of this metaclass contain zero or more *Package* metaclass instances but not *Classifier* instances (see Sect. 3.3); therefore DOMMLite forbids creation of classifiers outside of a package. In order to define a classifer one must enclose it inside a package.

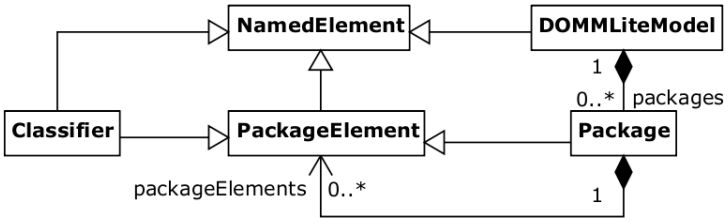

Fig. 2. Model organization

3.3. Classifier

The *Classifier* (see Fig. 3) abstract metaclass represents the superclass of metaclasses that describe key concepts of the language. Excluding *DataType*, these key concepts are, in DOMMLite terminology, also called complex types. There is a difference in the concept of a *Classifier* between the UML language and the DOMMLite language. For the sake of simplicity, the DOMMLite language unifies notions of type and classifier in the metaclass *Classifier* while those notions are separate in UML. There is no *Type* metaclass in DOMMLite – *Classifier* plays the role of the *Type* metaclass from the UML language. The classifier inherits the *PackageElement* and, consequently, can be nested inside packages. It is also a *NamedElement*, so all its descendants have a name, a short and a long description. The name should consist of letters, digits and underscores, and should begin with a letter, though these rules are not enforced by constraints in the current version of DOMMLite. The classifier name is usually mapped to the programming language identifier during model compilation; to make this mapping easier, these simple rules should be followed. The short description (*shortDesc*) is a description of an entity in a few words and is usually used as a classifier label in the GUI, or to aid model navigation and search. The long description (*longDesc*) can be arbitrarily long and is used to clarify the classifier's purpose. It can be used for tool tip generation, or for context-sensitive help support. The *NamedElement* metaclass, although it has different properties, represents the same concept from UML language.

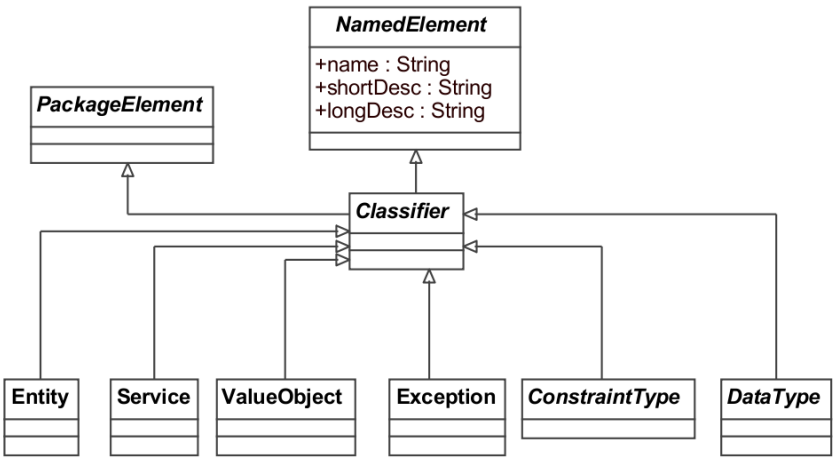

Fig. 3. The Classifier metaclass

3.4. Feature

Feature (see Fig. 4) is an abstract metaclass which represents properties (structural features) and operations (behavioral features) of main DOMMLite concepts with internal structure. *Feature* inherits the *TypedElement* metaclass, so all features have the following properties:

- ***required*** – If this flag is set, this feature's value must be defined. This serves as a support for NULL constraints in relational databases. For operations, this can be used as an indication that its return value is always non-null.
- ***many*** – If this flag is set, this feature is a collection (e.g. an array).
- ***ordered*** – This flag is taken into account only if the *many* flag is set to true. If it is set, this feature's values are ordered, meaning that elements of the collection have a notion of an index inside the collection.
- ***unique*** – If this flag is set, all elements of this feature must have a unique value. This flag is taken into account only if the many flag is set to true.
- ***multiplicity*** – The value of this property determines the multiplicity of the elements of multi-valued features. If it is set to zero, the multiplicity is not restricted. This flag is taken into account only if the *many* flag is set to *true*.
- ***type*** – The value of this property determines the type of this feature. The type of a feature can be any classifier, therefore the type can be simple or complex.

Features, being typed elements, can have a set of constraints (see Sect. 3.9), through which validation rules and tags can be attached to them for their further specification. Features are specialized into properties (the *Property* metaclass) and operations (the *Operation* metaclass).

3.4.1. Property

The *Property* (see Fig. 4) metaclass describes the structural features of the modeling elements. *Property*, being a typed element, can refer to other model elements which conform to the *Classifier* metaclass. Properties in DOMMLite can, on the semantic level, be compared to *EStructuralFeature* in ECore or *Property* in EMOF, but it is more similar to EMOF, because ECore divides this semantic concept into two metaclasses: *EAttribute* and *EReference*.

Property is a *TypedElement*; therefore it has type. If the type of a property is simple (primitive type or enumeration), we call it an attribute. If the type of a property is complex (e.g. *Entity*, *ValueObject* etc.), we call it a reference.

References in DOMMLite can be bidirectional. This ability is represented by the *oppositeEnd* reference of a *Property* metaclass, which "points to" the model element conforming to the *Property* metaclass. The *oppositeEnd*

reference must point to the model object conforming to the *Property* metaclass which is contained inside the model object referenced by the type of a reference. This is enforced by a constraint.

References can be used to represent a containment relationship (containment property) between model elements. The semantics of the containment relationship is similar to that of a composite association in UML or a containment relationship in ECore. Containment references affect the object's life-cycle. An object can not exist without its containing object.

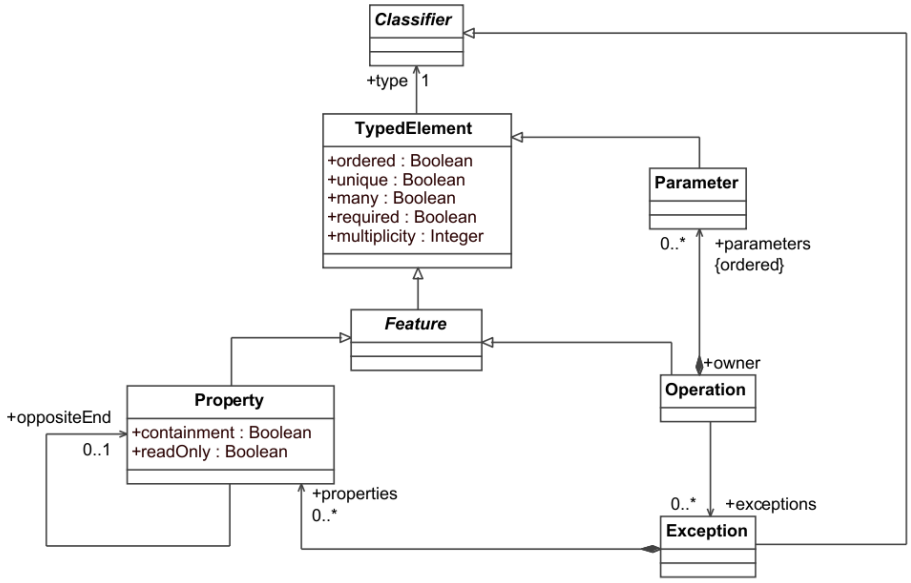

Fig. 4. TypedElement and Feature metaclass

3.4.2. Operation

The *Operation* (see Fig. 4) metaclass describes behavioral features of modeling elements. An operation can have parameters and a return type and can throw exceptions. The current version of DOMMLite deals primarily with structural properties of the observed systems, as a result of which modeling the behaviour of operations is currently out of scope of this language. As a means of supporting the specification of services (see Sect. 3.7) as well as the introduction of of behavioural model elements in latter versions of DOMMLite (see Sect. 8) we chose to introduce operations through operation signatures (name, return type, parameters, exceptions). The behavioural logic of operations is currently specified on the target platform programming language. The interpretation of operations and their semantics is left to the model compiler developer.

3.5. Entity

Entities (see Fig. 5) represent objects whose identity does not change throughout their lifetime. They have the ability to persist their state between application sessions. The identity of an entity is unique within the boundaries of a software system and is represented by one or more of its properties. Two entities are considered equal if their identities are equal.

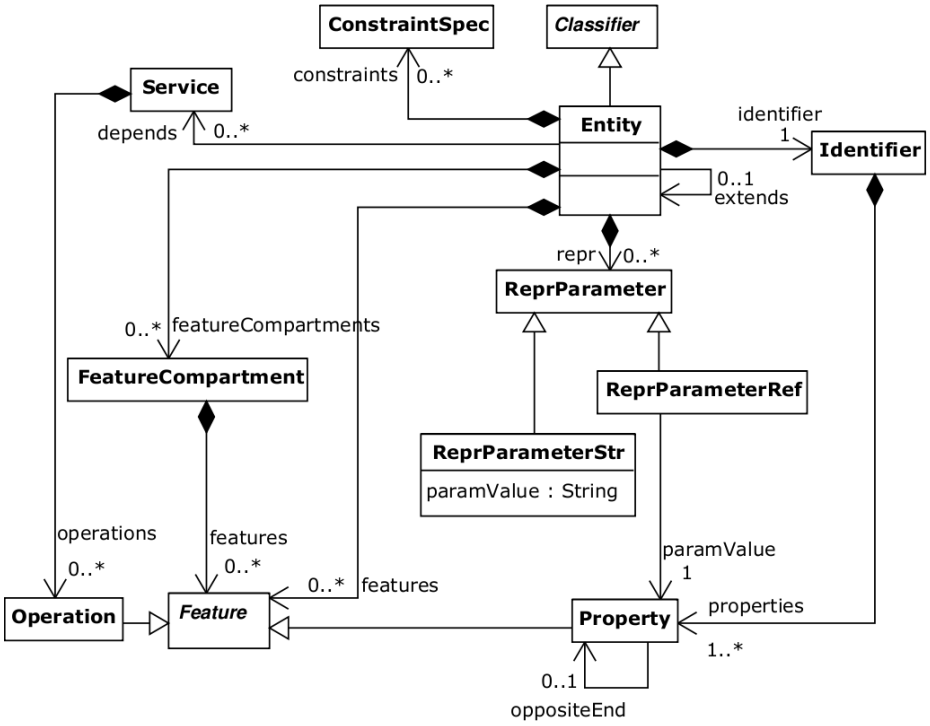

Fig. 5. Entity metaclass

3.5.1. Semantic Identifier

Identifier is a metaclass that represents the semantic identifier of an entity. Exactly one semantic identifier is defined for each entity. A semantic identifier consists of one or more properties which, together, uniquely identify an entity instance in the given software system.

In most cases, entities are persisted in relational databases, since they are still the most widespread storage mechanism. As a result, semantic identifier is closely related to the concept of a primary key. This concept deserves closer consideration.

Semantic vs. Synthetic identifier – There are two approaches in choosing properties which will represent the primary key of an entity in a relational database:

- ***Semantic identifier (natural key)*** – One or more existing properties of a business entity are chosen to represent the entity's primary key. We say that the identifier is semantic because it has meaning in the application domain. Choosing the right properties for an identifier is not an easy task, since the invariability of primary key value must be ensured throughout the lifetime of the entity instance.
- ***Synthetic identifier (surrogate key)*** – A synthetic identifier comprises properties without any business domain meaning. Its values are usually generated by the application or the database and its type is chosen based on the capabilities and performance of the database in dealing with such types. A synthetic identifier is a technical concept and does not need to be a part of the model.

Based on the experience in working with relational databases described in [27], the authors have come to the conclusion that using synthetic identifiers in relational databases is a better approach on the long run. On the other hand, using synthetic identifiers makes the implementation of the zoom technique [2] for manual data entry of references between entities (in some systems it is also known as the lookup technique) less efficient. The application user would have to specify the identifier of a referenced entity, which does not have any domain meaning. It would be irrational to expect that users would be able to remember these identifiers; this is why searching for referenced entities by the value of its other properties is the only viable option. The downside of this approach is that, in order to make a reference to another entity, the user would have to activate the search form for every reference that he or she makes. It is very common for the user to already know the semantic identifier of the referenced entity (e.g. social security number or bank account number); hence the need to call the search form each time a reference is entered, instead of directly specifying the semantic identifier, would severely slow data entry down.

Having that in mind, we propose a hybrid approach as a solution. On the modeling level a semantic identifier is always used. This will allow the use of approaches for automatic generation of the efficient zoom mechanism, which is used for reference entry and GUI forms navigation. In order to overcome shortcomings which the use of semantic keys in relational databases introduces, synthetic keys are generated for this purpose instead, while the application layer is in charge of mapping semantic identifiers to the synthetic database primary keys based on the information available in the DOMMLite model. Of course, DOMMLite does not impose use of synthetic identifier on the database level; it is only a recommendation. It is up to the developer of the source code generator to make a decision if the synthetic or the semantic identifier will be used in the database.

Global and local entity identifier – We classify unique entity identifiers into two categories: global and local. A local entity identifier is unique in the context of its containing entity; it does not need to be unique in the context of the entire modeled system. In DOMMLite local entity identifiers are modeled using *Identifier* metaclass. A global semantic identifier is unique in the context of the entire software system. It consists of the local semantic identifier of an entity combined with the global semantic identifier of the containing entity. If an entity does not have the containing entity, its local semantic identifier is equal to its global semantic identifier.

3.5.2. Feature Compartments

The basic idea of feature compartments is to logically group different features, both behavioral and structural, for easier model navigation inside coarse-grained entities, as well as to support automatic generation of GUI forms. In desktop applications compartments are usually used to create pages on the Tab visual component, which contains of visual components representing compartment properties. This enables us to generate more intuitive user interfaces without additional manual customization. For an example of feature compartment's usage see compartment *Contact Information* whose definition is shown in figure 10 and the generated web form is shown in figure 19.

3.5.3. Inheritance

Entities support single inheritance model. In the current version of DOMMLite inheritance is defined only at the level of the abstract and concrete syntax. Its semantics is left undefined and the model compiler developer is free to define its meaning. As a recommendation DOMMLite inheritance should follow the semantics of inheritance in other OO languages.

3.5.4. Service Dependency

Service dependency can explicitly be stated in the model. This information is used to support the *Dependency Injection* design pattern [28], thus making service reference available for entity operations by the time they get called without the need for the entity to obtain that reference by itself (e.g. for Spring framework this can be done by generating XML configuration files where reference injection is stated declaratively).

3.5.5. Entity Operations

An entity operation is described by the *Operation* metaclass (see Sect. 3.4.2). A business method that performs operations solely on one instance of an entity should be specified as an operation of that entity. It is recommended to model operations that operate on multiple instances of an entity as service operations.

3.5.6. Textual Representation

It is useful, if not necessary for entities to have a mechanism that will allow them to be represented in a textual form. For this purpose the collection *repr* of instances of *ReprParameter* metaclass is defined. The textual representation of an entity is obtained by concatenating strings (instances of *ReprParameterStr*) and textual representation of properties (instances of *ReprParameterRef*). This information is usually used for human-readable entity representation in the GUI. See figure 10 for an example (*firstName* and *lastName* are used for the textual represenation of the Student entity).

3.6. ValueObject

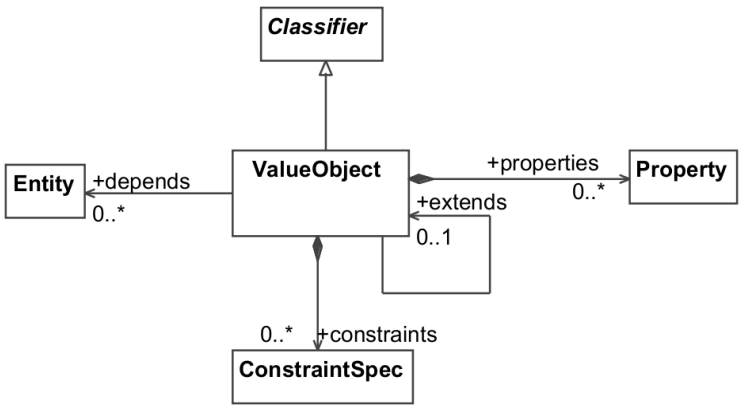

Fig. 6. ValueObject metaclass

Value (transfer) Objects (VO for short) are transient objects without identity. They are not meant to be persisted and are usually used to encapsulate data for interchange between different tiers of multi-tier applications. VOs can depend on entities (Fig. 6) if properties of a VO are based on properties of an entity. As with most metaclasses, the semantics of

VO can further be refined using a collection of constraints (see Sect. 3.9). Although VOs can have operations [14], in this version of DOMMLite language they are simple objects with structural features only. Inheritance of VOs is defined at the syntax level. Its precise semantics is left to the model compiler developer to define, but it is recommended to follow the semantics of inheritance defined in other OO languages.

3.7. Service

The role of the *Service* metaclass (see Fig. 7) is to describe objects whose purpose is to provide services to other domain objects. Services consist of logically interrelated operations that achieve a particular objective. These operations can query entities and can act on them by changing their state. In the current DOMMLite version services do not have properties, so their internal state is not modelled (they are stateless).

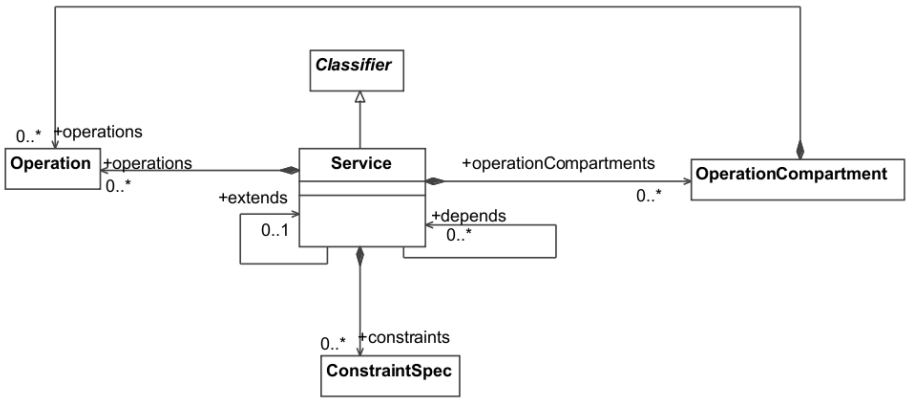

Fig. 7. Service metaclass

If necessary, service state can be modelled as an entity but the DOMMLite language currently has no support for explicitly stating which entity is used for state preservation of some particular service.

Service operations usually operate on multiple instances of entities or value objects. Operations that do not operate on a single entity instance, or do not operate on an entity instance at all should be modeled as service operations. Operations that query a single entity instance or change its state are usually better represented by entity operations (see Sect. 3.5). Service, being a *NamedElement*, has a name, a short and a long description. To facilitate model navigation and GUI generation, service operations can logically be grouped into operation compartments represented by the *OperationCompartment* metaclass.

Services can depend on each other. This information is used to implement the *Dependency Injection* design pattern. The single inheritance model is supported at the abstract and concrete syntax level. The semantics of

inheritance is left to the model compiler developer to define, but it is recommended to follow the semantics of inheritance from other OO languages. In the current implementation of the generator, service inheritance is not used (see Sect. 5). The semantics of services and service operations can further be refined using a collection of constraint specifications (see Sect. 3.9).

3.8. Validators and Tags

DOMMLite models can be augmented by validators and tags that can be attached to model elements (see Fig. 8). Validators define rules that should be enforced upon a running system in order to maintain a consistent state. For an example of validator usage see figure 10. Figure 19 shows a generated web form with applied validators.

Tags are simple constraints, similar to UML tags and stereotypes, which are used to alter or further refine the semantics of modeling elements. For example, the built-in tag *plural* is used to define the plural name of a modeling element, while the built-in tag *searchBy* is used to define properties of an entity that will be searched during keyword-based searches. The next section describes validators and tags in the context of extensibility.

3.9. Extensibility

The technique used to achieve extensibility (see Fig. 8) in DOMMLite is similar to that used in UML profiles. A modeler can introduce new data types, validator types and tag types into a DOMMLite model. These elements can be used in the rest of the model in the same way as the built-in ones.

Main metaclasses, which support extensibility, are *UserDataType* (see Sect. 3.1), *ConstraintType* and *ConstraintSpec* metaclasses. *ConstraintType* represents the definition of the type of a constraint, while *ConstraintSpec* is a concrete usage or instance of that type. *ConstraintType* defines language metaclasses to which a constraint can be applied (the *appliesTo** property). This information is used to constrain instances of *ConstraintType* (the modeling object that conforms to *ConstraintSpec*) so they can only be applied to the modeling element specified by the *appliesTo** property. The parameters collection of the *ConstraintTypeParameter* type defines the list of formal parameters of a constraint. Their types can be string, int (integer), ref (property reference) and ellipsis (variable number of parameters). *appliesTo* constraint as well as parameter types are checked during model editing and source code generation by a Check language rules. *ConstraintType* is specialized by *ValidatorType* and *TagType* metaclasses. *ValidatorType* metaclass represents the type of validators and the *TagType* metaclass represents the type of a tag, which can be instantiated in the model using *ConstraintSpec* metaclass.

There are two types of validators: *BuiltInValidatorType*, supplied with the DOMMLite language, and *UserValidatorType*, defined by the modeler at the model level. Figure 10 shows examples of built-in validators (*isOnlyDigits* and *isOnlyLetters*) as well as an example of a user-defined validator (*mod*). The *mod* validator accepts one parameter of the integer type. Validators are implemented on the target platform and the run-time form validation based on modelled validators is shown in figure 19. As we can see in figure 19, the source code generator (see Sect. 5) generates validators that, by default, do not pass validation (*isOnlyLetters* validator is undefined and therefore will not validate). After the implementation of validators on the target platform (see Fig. 18), it will be ensured that only valid data is stored in the database.

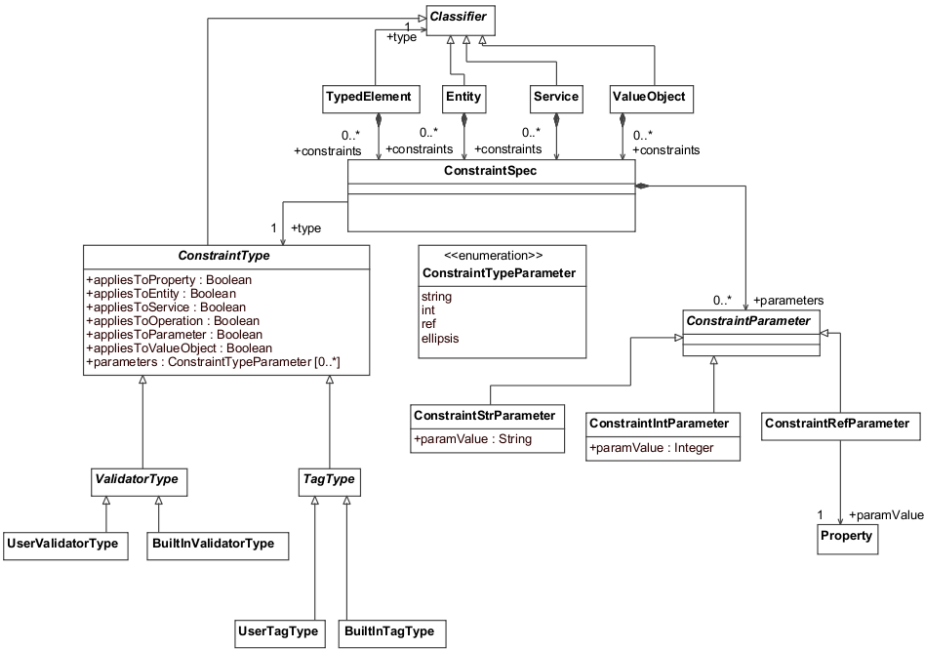

Fig. 8. Extensibility, validation and constraints

Tags can also be user-defined (*UserTagType*) or built-in (*BuiltInTagType*). Tags can be used to help identify a particular model element. For example, applying the finder tag to an entity operation could mark it as a so-called finder operation. Finder operation searches for and returns the entity or collection of entities that match given search criteria. Using this simple approach source code for this type of operation can automatically be generated.

The instance of *ConstraintType* is defined by the *ConstraintSpec* metaclass. Objects that conform to *ConstraintSpec* metaclass can be assigned to all typed elements, entities, services and value objects, as long as they respect constraints enforced by *appliesTo** property of their

ConstraintType metaclass. Constraint parameters must conform to the type and ordinal position of the constraint type specification. If the formal parameter of a constraint type specification is ellipsis, all its instances can use any number of parameters of any valid parameter type.

4. Concrete Syntax

Based on the abstract syntax different concrete syntaxes are possible. In this section we describe a textual concrete syntax implemented using xText language and tool, which is part of the openArchitectureWare generator framework [15]. xText is an EBNF-like language that can be used to specify textual concrete syntaxes as well as abstract language syntax (the metamodel) using the same definition. This feature of xText is used in the implementation of DOMMLite so that abstract syntax described in section 3 is defined along with the definition of DOMMLite concrete textual syntax. The rest of this section presents specifications of concrete syntaxes of two main language concepts: entity and service, as well as the syntax of extensibility support.

4.1. Entity Syntax

```
Entity:
	"entity" name=ID
		("extends" extends=[Entity])?
		("depends" depends+=[Service] ("," depends+=[Service])*)?
		(shortDesc=STRING)? (longDesc=STRING)?
	"{"
		identifier=Identifier
		("repr" repr+=ReprParameter ("+" repr+=ReprParameter)*)?
		("[" constraints+=ConstraintSpec ("," constraints+=ConstraintSpec)* "]")?
		(features+=Feature)*
		(featureCompartments+=FeatureCompartment)*
	"}";

Identifier:
	"ident" "{"
		(properties+=Property)+
	"}";

FeatureCompartment:
	"compartment" name=ID (shortDesc=STRING)? (longDesc=STRING)? "{"
		(features+=Feature)*
	"}";

ReprParameter:
	ReprParameterStr|ReprParameterRef;
ReprParameterStr:
	paramValue=STRING;
ReprParameterRef:
	paramValue=[Property];
```

Fig. 9. Entity syntax rule in xText language

Figure 9 shows the xText rule, which defines an entity. Definition of an entity begins with the keyword entity followed by the name of the entity. Entity inheritance relationship can be defined by the keyword *extends* followed by the name of the ancestor entity. Service dependency can be specified by the keyword *depends* followed by the list of comma-separated names of services this entity depends upon. An *Entity*, being a *NamedElement*, can have a short and a long description, which are defined as strings. Curly braces demarcate the entity body. An entity must define its semantic identifier. Semantic identifier definition starts with the keyword *ident* followed by a block demarcated by curly braces, and consists of a list of one or more properties. The definition of the string representation of an entity begins with the keyword *repr* followed by a list of strings or property names delimited by a plus sign. Constraints are specified inside square braces. Features are defined on the entity level or they can be grouped in feature compartments. Feature compartments begin with the keyword *compartment* followed by the compartment name and an optional short and long name.

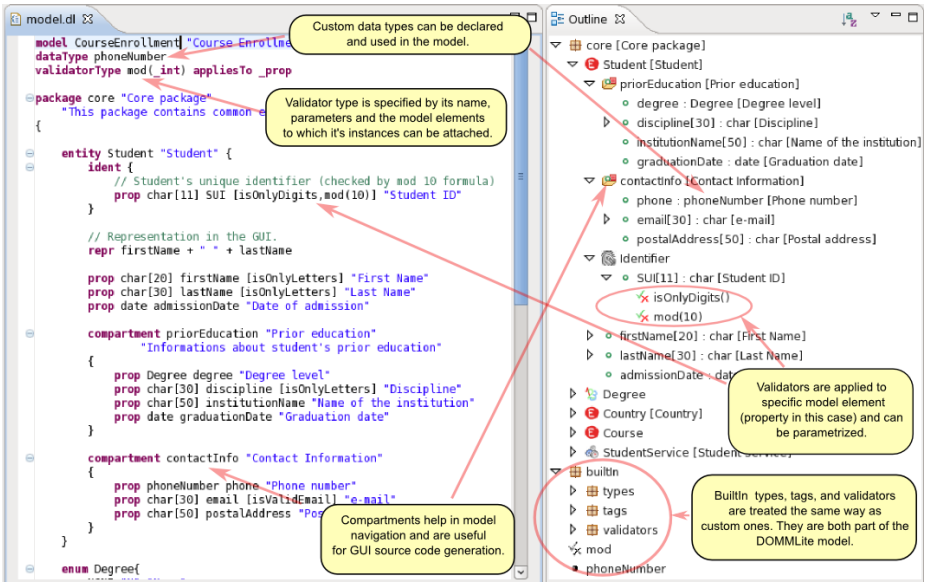

Fig. 10. Entity in the Eclipse-based DOMMLite model editor

Figure 10 represents an example of an entity *Student* in an Eclipse-based editor. The semantic identifier of the *Student* entity consists of the property *SUI (Student Unique Identifier*), which is numeric (validator *isOnlyDigits*), and its maximum length is set to 11. SUI is checked by a mod 10 formula using custom-defined validator *mod*. On the user interface this property will have the label *Student ID*. String rendering of the *Student* entity (*repr* keyword) will be done using first name and last name separated by a space. First and last names are constrained to contain letters only by a built-in validator

isOnlyLetters. There are two compartments: *priorEducation* and *contactInfo*. The *priorEducation* compartment has a long description defined in it, which will be used to further explain the content of the compartment. The *phoneNumber* data type, used in the *contactInfo* compartment and shown in the outline view, is user-defined. It is defined at the model level using the *dataType* keyword (see Sect. 3.9).

4.2. Service syntax

Service's xText sintax rule is shown in Figure 11. Service definition begins with the keyword *service* followed by the service name. Service inheritance is defined by the keyword *inherits* followed by the name of the ascendant service. Dependency is defined by the keyword *depends* followed by a comma-separated list of service names. Service body contains the specification of constraints, service operations and operation compartments.

```
Service:
      "service" name=ID
      ("extends" extends=[Service])?
      ("depends" depends+=[Service] ("," depends+=[Service])*)?
      (shortDesc=STRING)? (longDesc=STRING)?
      "{"
            ("[" constraints+=ConstraintSpec ("," constraints+=ConstraintSpec)* "]")?
            (operations+=Operation)*
            (operationCompartmens+=OperationCompartment)*
      "}";

OperationCompartment:
      "compartment" name=ID (shortDesc=STRING)? (longDesc=STRING)? "{"
            (operations+=Operation)*
      "}";
```

Fig. 11. Service syntax rule in xText language

4.3. Extensibility Features Syntax

Figure 12 shows the syntax rules for extensibility support. User defined data types are defined by the keyword *dataType* followed by the name of the new type.

TagType and *ValidatorType* have similar syntaxes. Their definition begins with the keyword *tagType* (or *validatorType*) and the name of the tag (validator). The name is followed by the definition of parameters as a comma-separated list of type parameters. After the keyword *appliesTo* metaclasses should be defined to which this tag (validator) can be applied. This information is used by the model editor as well as the model compiler to perform model validation.

The same type of constraint can be applied, if defined in that way, to different types of modeling constructs. Formal parameters of the constraint

type definition represent the types of real parameters in constraint usage specification. The type of a formal parameter can be: _string - string surrounded by quotation marks, _int - integer, _ref - reference to a property represented by its name, ... - ellipsis means that parameter types and their number is undefined.

```
UserDataType:
      "dataType" name=ID (shortDesc=STRING)? (longDesc=STRING)?;

UserTagType:
      "tagType" name=ID
            ("("(parameters+=ConstraintTypeParameter)?
            ","parameters+=ConstraintTypeParameter)* ")")?
            ("appliesTo"
                  ((appliesToEntity?="_entity") |
                  (appliesToProperty?="_prop") |
                  (appliesToParameter?="_param") |
                  (appliesToOperation?="_op") |
                  (appliesToService?="_service") |
                  (appliesToValueObject?="_valueObject"))*)?
            (shortDesc=STRING)? (longDesc=STRING)?;

UserValidatorType:
      "validatorType" name=ID
            ("("(parameters+=ConstraintTypeParameter)?
            ","parameters+=ConstraintTypeParameter)* ")")?
            ("appliesTo"
                  ((appliesToEntity?="_entity") |
                  (appliesToProperty?="_prop") |
                  (appliesToParameter?="_param") |
                  (appliesToOperation?="_op") |
                  (appliesToService?="_service") |
                  (appliesToValueObject?="_valueObject"))*)?
            (shortDesc=STRING)? (longDesc=STRING)?;

Enum ConstraintTypeParameter:
      string="_string"|
      int="_int" |
      ref="_ref" |
      ellipsis="..."
;
```

Fig. 12. Data types and constraint types syntax rules

Figure 13 show the syntax rule for constraint usage. A constraint is defined by its name followed by real parameters surrounded by a pair of braces. In order to be used, a constraint must have its type defined in the model as a built-in or user-defined constraint type or there will be errors during model validation. The type and the number of real parameters must conform to the constraint type formal parameter specification.

```
ConstraintSpec:
       type=[ConstraintType] ("(" (parameters+=ConstraintParameter)?  ("," 
parameters+=ConstraintParameter)* ")")?;

ConstraintParameter:
      ConstraintIntParameter|ConstraintRefParameter|ConstraintStrParameter;

ConstraintStrParameter:
      paramValue=STRING;

ConstraintIntParameter:
      paramValue=INT;

ConstraintRefParameter:
      paramValue = [Property];
```

Fig. 13. Constraint usage specification syntax rules

5. Source Code Generator

Details of DOMMLite execution semantics are given in the form of a source code generator (model compiler) for the chosen target platform. For the purpose of building the prototype proof-of-concept implementation, Django web framework [29], which is based on Python programming language as our target platform, was chosen.

```
«DEFINE adminClass FOR Entity»
class «name»AdminForm(forms.ModelForm):
      class Meta:
            model = «name»
      «FOREACH allPropertiesWithKeyFields() AS prop-»
            «IF !prop.allConstraints().isEmpty-»

      def clean_«prop.name»(self):
            «prop.name» = self.cleaned_data['«prop.name»']
            «FOREACH prop.eContents.typeSelect(ConstraintSpec)
                  .select(e|ValidatorType.
                        isAssignableFrom(e.type.metaType)) AS validator-»
            «validator.type.name»(«prop.name»«IF !
                        validator.parameters.isEmpty»,«ENDIF»
            «EXPAND validatorParam FOREACH validator.parameters»)
            «ENDFOREACH-»
            return «prop.name»
            «ENDIF-»
      «ENDFOREACH-»
..... code removed
«ENDDEFINE»
```

Fig. 14. Xpand template fragment for Django admin class generation

The model compiler is defined as a set of templates based on Xpand language, a powerful DSL for defining templates for code generation, which is a part of the openArchitectureWare framework. Figure 14 shows a fragment of the Xpand template used to generate a Django admin form class for every entity in the DOMMLite model.

Figure 15 shows a fragment of a generated, in Django terminology, model. A Django model is generated for every entity in the DOMMLite model and it contains meta-data used for creating tables in the database and for configuring the Django Object-Relational Mapper (DORM). We can see that built-in types are mapped to Django model fields (e.g. for the *firstName*, which is of type *char*, a model field of type *models.CharField* will be generated). Custom types are mapped to custom fields that are specified manually in the *custom_fields* python module (e.g. for the phone field, which is of the user-defined type *phoneNumber*, a model field of type *custom_fields. PhoneNumberField* will be generated).

```
class Student(models.Model):
      # ------------------------------ Identifier -----------------------------
      SUI = models.CharField(verbose_name=u"Student ID",
                             unique=True,max_length=11)
      # -----------------------------------------------------------------

      firstName = models.CharField(verbose_name=u"First Name",
                                   blank=True,max_length=20)
      lastName = models.CharField(verbose_name=u"Last Name",
                                   blank=True,max_length=30)
      admissionDate = models.DateField(verbose_name=u"Date of admission",
                                   null=True, blank=True)

      #....... part of code removed ....
      # ------------------    Compartment contactInfo    ----------------------
      phone = custom_fields.PhoneNumberField(verbose_name=u"Phone number",
                                   null=True, blank=True)
      # ....  part of code removed ....
```

Fig. 15. Fragment of a generated Django model class

In figure 19 we can see that the phone field will be properly validated. *PhoneField* can be used in multiple places throughout the model and the generator will take care to create the right django model fields, which will instantiate the manually created *custom_fields.PhoneNumberField* class. Therefore, the generator doesn't need to be altered for user-defined types. Django will create a synthetic primary key (a column called id usually of the auto-incrementing type if the database supports it) that will be used in the framework for model/entity identification. This synthetic identifier can be accessed by the DORM API and used in service and entity operations.

In figure 16 a fragment of a generated django admin form class is presented. Django uses admin classes to drive the admin application, which is capable of generating CRUD forms on-the-fly from the information supplied in the Django model and admin classes (which are generated in this case). Admin form classes will call validators specified in the DOMMLite model to

ensure that field values are validated prior to their storing in the database. Generated validators (Fig. 17) by default fail for every field value.

```
class StudentAdminForm(forms.ModelForm):
      class Meta:
            model = Student

      def clean_SUI(self):
            SUI = self.cleaned_data['SUI']
            isOnlyDigits(SUI)
            mod(SUI,10)
            return SUI
```

Fig. 16. Fragment of a generated Django admin class

```
#PROTECTED REGION ID(mod) START
def mod(value, param0_int):
      raise ValidationError(_(u'Validator "mod" is applied to
                              this field but is not defined yet.'))
#PROTECTED REGION END
```

Fig. 17. Generated default implementation of mod validator

After validator implementation (Fig. 18), the Django admin form will do proper validation of an entered field value (Fig. 19). In this case, validators are implemented as protected regions (a feature of oAW generator) to preserve manual modifications during subsequent generator invocations.

```
#PROTECTED REGION ID(mod) ENABLED START
def mod(value, modulus):
    _sum = 0
    alt = False
    for d in reversed(str(value)):
      d = int(d)
        if alt:
            d *= 2
            if d > modulus-1:
                d -= modulus-1
        _sum += d
        alt = not alt
    if not (_sum % modulus) == 0:
      raise ValidationError('Number does not pass mod(%d) test.' % modulus)
#PROTECTED REGION END
```

Fig. 18. Validator mod manually implemented in Python language

The execution of the Django admin application generates during run-time execution Web based forms (Fig. 19) for basic CRUDS operations, without any manual modifications of the Django application generated from the DOMMLite model. However, more complex business operations and workflows must be implemented on the target platform, as they still cannot be specified in the DOMMLite language.

The skeletons for entity and service operations can also be generated as protected regions also or other capabilities of target platform can be used for mixing generated and manually written source code (e.g. inheritance, or function override in case of Python) [26].

In the presented generator services are mapped to Python modules and service operations are mapped to module functions. Dependency information between services and entities is used to generate the proper import section, which will only import those Django models (DOMMLite entities) that service depends upon. Operations are implemented manually in the Python programming language using the Django framework.

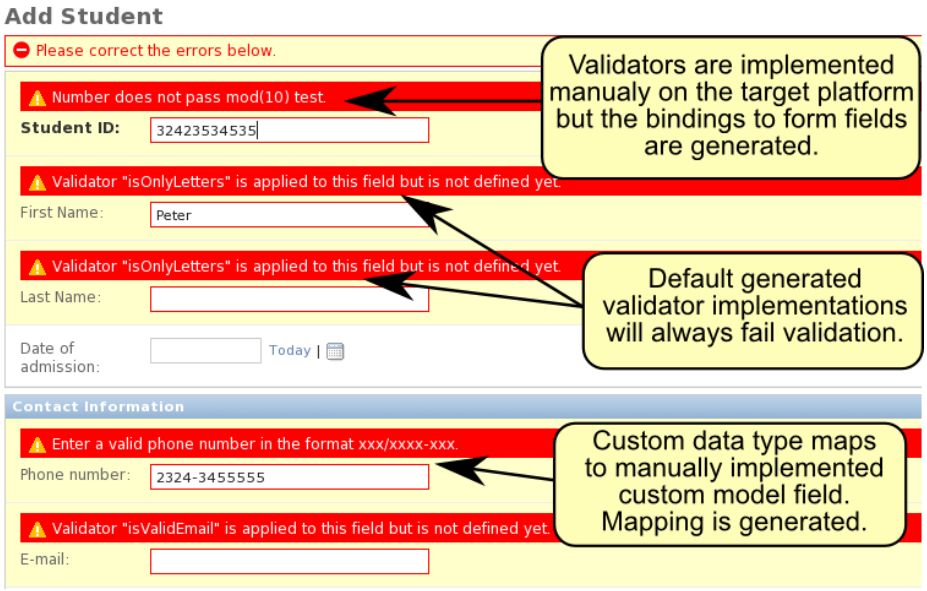

Fig. 19. Django entry form for Student entity with custom validator and field type

6. Eclipse Editor for DOMMLite Models

Using the specification of a language in the form of xText rules as a starting point, openArchitectureWare is capable of generating an almost fully functional Eclipse-based model editor (Fig. 10). In order to achieve full functionality, additional configuration and completion of the generated editor should be performed.

First, we have to define modeling constraints, which will be enforced during model editing as well as during model compilation. For this purpose oAW

offers a DSL called Check. Figure 20 shows the Check rule, which ensures that constraint specification is applicable to the given modeling element.

```
context ConstraintSpec ERROR "Constraint " + type.name + " is not applicable to
this model element!" :
      (eContainer.metaType==Entity && type.appliesToEntity) ||
      (eContainer.metaType==Service && type.appliesToService) ||
      (eContainer.metaType==Property && type.appliesToProperty) ||
      (eContainer.metaType==Operation && type.appliesToOperation) ||
      (eContainer.metaType==Parameter && type.appliesToParameter) ||
      (eContainer.metaType==ValueObject && type.appliesToValueObject)
;
```

Fig. 20. Check rule for constraint specification

In order to provide support for the outline view, there are operations that need to be supplemented using the oAW functional language Extend. These operations are *label* - for supplying the node label in run-time, and *image* for supplying node icon. The outline rule for entity is given in figure 21. The outline of a model in Eclipse is presented on the right side of figure 10.

```
String image(emf::EObject this) :
            metaType.name.split("::").get(1) + '.png';

label(dommlite::Entity this) : defaultExtendable(this);

String defaultExtendable(Object this):
      metaType.getProperty("name")!=null ?
            metaType.getProperty("name").get(this).toString() +
            (metaType.getProperty("extends").get(this)!=null ?
                  "<"+((dommlite::NamedElement)metaType.getProperty("extends")
                        .get(this)).name : "") +
                  (((List)metaType.getProperty("depends").get(this)).size>0 ?
                  "("+((List)metaType.getProperty("depends")
                        .get(this)).collect(e|((NamedElement)e).name)
                              .commaSeparated()+")" : "") + shortDesc(this) : null;
```

Fig. 21. Outline rules for the Entity metaclass

```
List[Proposal] completeProperty_oppositeEnd(emf::EObject ctx, String prefix) :
      ((Property)ctx).type.eContents.typeSelect(Property).select(e|
            e.type==((Property)ctx).containingEntity()).
            collect(x|newProposal(x.label(),x.id(),x.image()));
```

Fig. 22. Extend function for supporting bidirectional reference code completion

Code completion is supplemented by the implementation of the *complete** function. Code completion rule for bidirectional references is shown in figure 22. The rule from figure 22 will ensure that only properties from *Classifier* on the other side of relation that reference *Classifier* on this side of the relationship will be offered during code completion.

7. Related Work

Work related to the topics discussed in this paper includes research on design and application of domain-specific modeling languages and domain-specific languages in general to different domains.

DSLs have been most successful and are particularly popular in the domain of embedded systems. In [30] an approach to building embedded component infrastructures is proposed, which is based on the combination of component/container infrastructures (including the underlying communication middleware) with model-driven software development techniques. An example of a DSL for specifying an embedded application (a simple weather station) using MDSD is analyzed. The example metamodel is implemented as an extension of the UML metamodel. DSL concrete syntaxes are UML and XML based. In [31] a PICML DSML is proposed which simplifies and automates many activities associated with developing, and deploying component-based Distributed Real-time Embedded systems. In particular, PICML provides a graphical DSML-based approach which is used to define component interface definitions, specify component interactions, generate deployment descriptors, define elements of the target environment, associate components with these elements, and compose complex DRE systems from these basic systems in a hierarchical fashion. [32] presents a Kiosk Application Generator (KAG), a generator for Kiosk Applications featuring its own templating language for the description of the generated code, as well as a declarative DSL for form-flow based application description. Graphics Adaptor Language (GAL) [33] is a DSL for the generation of video display device drivers.

Integration of tools, data and services is another domain where DSLs have been extensively used. In [34] authors investigate the application of DSLs in the area of tool integration. They have defined a Tool Integration Framework based on the concept of Domain Schema, which is specified as a Model Specification File (MSF) written in a declarative DSL that captures the data model for the various entities and their relationships within a tool. Since it is a data-structure description language, it is in many respects similar to DOMMLite. However, its domain of application is quite different. DSL for MSF, with an aim to be the least-common denominator for different kinds of tools, has a much simpler abstract and concrete syntax. Another application of DSLs for integration purposes is presented in [35]. In this paper the Enterprise-Application Integration approach is described based on predefined components configured with DSL programs (Domain-Abstract Representation). The concrete syntax used for DAR is XML based.

The ideas, which have been used in DOMMLite to support the generation of standardized GUI forms, can be found in [2,3,4,5]. The tool for rapid prototyping of large-scale business information systems presented in [2] uses UML as a design language. The UML model is then transformed to metadata kept in the Application repository. Meta-data in the Application repository is customized by a Form Generator tool, which utilizes this information to generate application source code. Meta-data in the Application repository,

although kept in a database and edited by a special-purpose tool, can be considered to be a DSL for the description of GUI forms. Using DOMMLite instead of UML makes this additional step of transforming UML to Application repository and doing further customization unnecessary, since all metainformations needed by a GUI form generator can be specified in the DOMMLite model itself.

Intermediate Form Representation (IFR) [5] is an XML-based DSL for the description of user interface forms, which supports multiple environments for user interface implementation. IFR files can be customized in terms of functionality and layout, while multiple iterations of code generation preserve all manual customizations done in between iterations. IFR files are generated from the Enterprise Java Beans (EJB) data model using Java introspection, making it an interesting alternative in the case of reengineering already existing EJB-based solutions.

In [36] the author investigates using UML as a basis for DSL construction through implementation of a DSL for modeling applications, targeting an already existing business application framework. The language described, although similar in concepts to DOMMLite, lacks support for in-model extensibility and coarse-grained entities (see Sect. 3.5.2), design time validation, run-time validation. As concluded by the author, UML, being a general-purpose modeling language, is not very well suited for the construction of DSLs.

AndroMDA [37] is an open-source MDA framework capable of generating source code for different platforms out of models specified in the UML. A model is defined using UML with stereotypes and exported to XMI format. From there a maven-driven build process parses it and the source code is generated by executing so-called cartridges (modules that perform code generation).

The language most alike DOMMLite, in terms of concepts and tools used, is Sculptor [38]. Sculptor is developed as a part of the open-source Fornax platform[8]. It is actively developed and documented, and is capable of generating source code for desktop RCP and Web applications with support for different technologies such as Hibernate, Spring, EJB, JSP, JFC etc. which is a very good option for teams that need to target different platforms and do not have time for development of source code generators. Being an open-source project, it also brings high quality of generated source code, as the templates are peer reviewed by all contributors and users. Sculptor follows Domain-Drive Design concepts very closely. As of this writing there seem not to be any published scientific papers describing Sculptor in more detail. In comparison to DOMMLite it features out-of-the-box source code generators for different popular technologies but at this time it does not have support for coarse grained entities (see Sect. 3.5.2) nor in-model extensibility (see Sect. 3.9).

8 http://www.fornax-platform.org/

8. Conclusions and Future Work

In this paper we have presented an extensible DSL implemented using MDE ideas and techniques, which is well suited for database-oriented business applications. From a DOMMLite model an entire application with navigation, and CRUDS operations can be generated. By using highly abstract languages that are based on concepts from the domain at hand, significant productivity increase can be achieved. The code generator propagates any change made in the model to all generated files, making the implementation consistent with the model and eliminating errors due to human factors, which inevitably occur if these changes are done manually. We have anticipated code quality improvement, since coding guidelines for generated code are enforced by templates and improvements in templates are immediately reflected on all generated code.

Hand-written code still needs to be maintained manually, which may lead to errors caused by misalignment to the generated code. However, using statical code analysis offered by contemporary integrated development environments, as well as test-driven development techniques, many bugs can be identified and eliminated quickly.

DOMMLite is designed from the ground up to be simple to use and simple to extend. By implementing its extension mechanism, it is possible to extend the language semantics on the model level without changing the metamodel and source code generator. This mechanism is fully supported by the eclipse editor with code completion and constraint validation for newly defined constraint types.

The development of DSLs and DSMLs is nowadays significantly simplified with the appearance of generator frameworks such as openArchitectureWare, which permit languages to be designed and supporting editors and compilers to be prototyped more quickly.

Further research and development is focused on extending the language and the functionality of the supporting tools. We plan to: (1) – extend the language to support behavioral elements (operation semantics), (2) – define the visual syntax of the language, as well as a supporting graphical editor, (3) – support additional target platforms, (4) – implement support for model version control, (5) – implement support for model and language evolution.

References

1. openArchitectureWare Generator Framework. [Online] Available: http://www.openarchitectureware.org/ (current December 2008)
2. Milosavljević, G., Perišić, B.: A method and a tool for rapid prototyping of large-scale business information systems, Computer Science and Information Systems ,vol. 1, pp. 57-82 (2004)

3. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User interface code generation for data-intensive applications with ejb-based data models, in Software Engineering Research and Practice (SERP'03), Las Vegas, NV, (2003)
4. Milosavljević, B., Vidaković, M., Konjović, Z.: Automatic Code Generation for Database-Oriented Web Applications, pp. 89-97. Recent Advances in Java Technology: Theory, Application, Implementation, Trinity College Dublin (2003)
5. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User interface code generation for ejb-based data models using intermediate formrepresentations, in ACM Principles and Practice of Programming in Java,(Kilkenny, Ireland) (2003)
6. Schmidt, D. C.: Guest editor's introduction: Model-driven engineering, Computer, vol. 39, no. 2, pp. 25-31 (2006)
7. Tolvanen, J.-P., Sprinkle, J., Gray J.: The 6th oopsla workshop on domain-specic modeling, in OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications , (New York, NY, USA), pp. 622-623, ACM (2006)
8. Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., Selic, B.: An MDA manifesto, MDA Journal (2004)
9. Dejanović, I.: Meta-model, model editor and business application generator, (Master's thesis). Author's reprint, Library of Faculty of Technical Sciences, Novi Sad, Serbia (2008)
10. Bézivin, J., Jouault, F., Kurtev, I., Valduriez , P.: Model-based DSL frameworks, Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, pp. 22-26 (2006)
11. OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.1.2, Final Adopted Specication, OMG Document formal/2007-11-04 (2007)
12. Meta Object Facility (MOF) Core Specication. Version 2.0 (2006)
13. Eclipse Modeling Framework - EMF. Online, Available: http://www.eclipse.org/modeling/emf/ , (current December 2008)
14. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software . Addison-Wesley Professional (2004)
15. Efftinge S., Völter, M.: oAW xText: A framework for textual DSLs, in Eclipse Summit 2006 Workshop: Modeling Symposium (2006)
16. Van Deursen, A., Visser, J.: Domain-specic languages: an annotated bibliography, ACM SIGPLAN Notices , vol. 35, pp. 26-36 (2000)
17. Wile, D. S.: Supporting the dsl spectrum, JOURNAL OF COMPUTING AND INFORMATION TECHNOLOGY , vol. 9, pp. 263-288 (2001)
18. Hatton, L.: The t experiments: Errors in scientic software, IEEE COMPUTATIONAL SCIENCE & ENGINEERING , vol. 4, no. 2, pp. 27-38 (1997)
19. Mernik, M., Sloane, A. M.: When and how to develop domain-specic languages, ACM Computing Surveys (CSUR) , vol. 37, pp. 316-344 (2005)
20. Fowler, M.: Domain specic languages. Online, Available: http://martinfowler.com/dslwip/ (current December 2008)
21. Cunningham H. C.: A little language for surveys: Constructing an internal dsl in ruby, ACM-SE 46, Auburn, Alabama, USA (2008)
22. Paul, R.: Designing and implementing a domain-specic language, LinuxJ. , vol. 2005, no. 135, p. 7 (2005)
23. Kabanov, J., Raudjärv, R.: Embedded typesafe domain specic languages for java, pp. 189-§197, ACM New York, NY, USA (2008)

24. Hudak , P.: Building domain-specic embedded languages, ACM Computing Surveys (CSUR) , vol. 28, p. 196 (1996)
25. Haugen, Ø., Mohagheghi, P.: A Multi-dimensional Framework for Characterizing Domain Specific Languages, Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Computer Science and Information System Reports (2007)
26. Völter, M., Stahl, T.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
27. Bauer, C., King, G.: Hibernate in Action, Manning Publications (2004)
28. Fowler, M.: Inversion of control containers and the dependency injection pattern. Online, Available: http://www.martinfowler.com/articles/injection.html (current February, 2008) (2004)
29. Django web framework. Online, Available: http://www.djangoproject.com/, (current December, 2008).
30. Vöelter, M., Salzmann, C., Kircher, M.: Model driven software development in the context of embedded component infrastructures, LECTURE NOTES IN COMPUTER SCIENCE , vol. 3778, p. 143 (2005)
31. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D. C.: A platform-independent component modeling language for distributed real-time and embedded systems, Journal of Computer and System Sciences , vol. 73, pp. 171§-185 (2007)
32. Živanov, Ž., Rakić, P., Hajduković, M.: Using code generation approach in developing kiosk applications, Computer Science and Information Systems , vol. 5, pp. 41-§59 (2008)
33. Thibault, S. A., Marlet, R., Consel, C.: Domain-Specic Languages: From design to implementation application to video device drivers generation, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, pp. 363§-377 (1999)
34. J. Gray and G. Karsai, An examination of dsls for concisely representing model traversals and transformations, p. 10 (2003)
35. Nussbaumer, M., Freudenstein, P., Gaedke, M.: The impact of domain-specific languages for assembling web applications, Engineering Letters, vol. 13 (2006)
36. Anonsen, S.: Experiences in modeling for a domain specic language, UML Modeling Languages and Applications , vol. 3297/2005, pp. 187§-197 (2005)
37. AndroMDA - MDA framework. Online, Available: http://www.andromda.org, (current December 2008)
38. Enterprise java community: Improving developer productivity with sculptor. Online, Available:http://www.theserverside.com/tt/articles/article.tss?l=ProductivityWithSculptor, (current June 2007) (2007)

Igor Dejanović received his M.Sc. (5 years, former Diploma) degree from the Faculty of Technical Sciences in Novi Sad. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he works as a teaching assistant at the Faculty of Technical Sciences at the University of Novi Sad, where he assists in teaching several Computer Science and Software Engineering courses. His research interests are related to Domain-Specific Languages, Model-Driven Engineering and Software Configuration Management.

Gordana Milosavljević is a teaching assistant and Ph.D. student at University of Novi Sad, Faculty of Engineering, Computer Sciences Department. She has received her B.Sc. and M.Sc. from University of Novi Sad, Faculty of Engineering, Computer Sciences Department. Her research interests focus on software engineering methodologies, rapid development tools and enterprise information systems design.

Branko Perišić is an associated professor at University of Novi Sad, Faculty of Technical Sciences. He has received his engineer diploma from University of Sarajevo, Faculty for electrical engineering, M.Sc. and PhD diplomas from University of Novi Sad, Faculty of Technical Sciences. He is currently a Computer center manager and leads Software development team at Faculty of Technical Sciences. As a teaching professor he has developed and teached a variety of Computer Engineering, Software Engineering and Information System Design courses at different Universities. His major research interests are related to Model Driven Software Development, Business Information Systems Design, Software Configuration Management and Secure Software Design.

Maja Tumbas is a teaching assistant at the Faculty of Technical Sciences at the University of Novi Sad, where she received her M.Sc. degree. Her fields of interest include different areas of software engineering, including software modeling and computer security.

Received: February 03, 2009; Accepted: December 18, 2009.