Metamodel-based UML Notations for Domain-specific Languages

Achim D. Brucker and Jürgen Doser

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker, doserj}@inf.ethz.ch

Abstract We present a metamodel-based approach for specifying UML notations for domain-specific modeling languages. Traditionally, domain specific languages are either defined by UML profiles or using metamodels. We provide a generic integration of these two methods supporting arbitrary UML profiles and metamodels. Our approach provides a bi-directional mapping between the UML notation and the metamodel of the domain specific language. We use OCL constraints that are embedded into the metamodel, for describing the mapping between the UML notation and the metamodel.
Moreover, we describe an implementation, as ArgoUML-plugin, for arbitrary SecureUML dialects.
Key words: DSL, UML, OCL, UML Profile, Metamodel, MOF, SecureUML

1 Introduction

The success of general-purpose modeling languages, especially the Unified Modeling Language (UML), does not render Domain-specific Modeling Languages (DSL) superfluous. On the one hand, UML provides a rich tool support, ranging from requirements engineering, over modeling to code generation; on the other hand, it is clumsy for tasks that can profit from integrating domain-specific restrictions. Moreover, large languages like UML usually lack a detailed, formal semantics needed for formal analysis. Domain-specific languages are often small and thus provide a good basis for domain-specific formal analysis and fully-automated tool support. In addition, DSLs usually utilize the notation domain experts are used to, which increases their acceptance. As DSLs are very specific, however, they usually do not benefit from the wealth of tools available for general-purpose modeling languages.

A common idea for combining the advantages of general-purpose modeling languages and domain specific languages is to define a DSL in terms of a general-purpose modeling language, e. g., UML. Such a UML-based domain-specific language can restrict the UML, for example, to certain model-elements and also introduce, by using stereotypes, new "types" into the language.

Classically, there are two ways for defining UML-based domain-specific modeling languages. First, domain-specific languages can be defined as a UML profile in a lightweight way, using stereotypes and tagged values. Second, the Meta-Object Facility (MOF) [10] can be used to either extend the UML metamodel,

4th International Workshop on Language Engineering (ATEM 2007), pp. 1–??, 2007.

or to directly define the metamodel of the new modeling language without any dependency on UML.

As most current UML tools support the definition of custom stereotypes and tagged values, the first approach has the advantage that most UML tools can readily be used to apply this approach. Some UML tools even allow to change the *presentation* (e. g., using different colors, or icons) based on the UML profile. While this approach by definition clearly defines the *concrete syntax* for the DSL at hand, the abstract syntax of it is at most only implicitly defined, and some aspects of the abstract syntax cannot be conveniently defined at all.

In contrast, the second approach clearly defines the abstract syntax. A drawback of the second approach is that it may require one to extend the CASE tool itself, in particular the storage components, i. e., the repository, and the visualization components, to support the DSL metamodel.

A naive combination of both approaches is frequently done by defining separately the abstract syntax by a metamodel, and the concrete syntax by a UML profile. This does not provide a well-defined mapping from the UML models the developer works with, to instances of the DSL metamodel that define the meaning of these models.

In this paper, we present a new approach for defining DSLs that combines the advantages of the previous two approaches, while avoiding the aforementioned problem. The fundamental idea of our approach is to encode a UML profile, and thus the concrete syntax of the DSL, into the metamodel of the DSL. This encoding provides a bi-directional mapping between the concrete syntax and the abstract syntax of the DSL. Overall, our approach allows for consistently defining the concrete syntax, the abstract syntax and the semantics of a DSL.

Moreover, we propose an architecture for implementing DSL support for standard UML CASE tools that only requires a programming interface to the model repository of the CASE tool. In particular, the model repository of the CASE tool does not need to support new metamodels. This architecture was used for implementing a plug-in for the CASE tool ArgoUML. This plugin supports SecureUML, a domain-specific modeling language for access control requirements.

The Plan of the Paper. After a brief introduction into domain specific languages in general and SecureUML in particular, we present in Section 3 the key concepts of integrating concrete UML notations into metamodels. In Section 4 we propose a concrete architecture for implementing DSL-specific extensions of CASE tools and in Section 5 we discuss the advantages and disadvantages of our approach in detail. Finally, in Section 6 we summarize related work and draw conclusions in Section 7.

2 Background

2.1 Domain Specific Languages

A Domain Specific Language (DSL) should define both syntax *and* semantics of its domain of application. In particular, a DSL may be an orthogonal extension

to UML instead of only a restriction of existing UML concepts. Especially in the context of UML, two mostly distinct methods for defining DSLs are used: DSLs are either defined by a metamodel or a UML profile (see Table 1 for a brief comparison of both approaches).

	Metamodel	UML Profile
new attributes	yes	no
new default datatype instances	yes	no
new Associations	yes	no
new methods	yes	no
new types	yes	no
adding subtypes	yes	yes
deleting/renaming UML types	no	no
any UML tool	no	yes
concrete syntax	no	yes
abstract syntax	yes	no
semantic definition	easy	difficult

Table 1. Comparing Metamodel-based vs. Profile-based DSL Definitions

In principle, defining a UML profile means to introduce a set of new UML stereotypes. Thus, a UML profile is a light-weight way of classifying model elements. Of course, defining a DSL only by stereotypes is a very limited way to describe the syntax of the DSL and giving a semantics for a UML profile is very hard. Nevertheless, this approach for defining DSLs is widely used as it is supported by many CASE tools.

Defining a DSL by defining a metamodel allows for defining the DSL, independently from UML, in terms of its datatypes, methods, etc. Moreover, defining the (formal) semantics of a DSL based on its abstract syntax is usually easier than describing the semantics for the concrete syntax. Sadly, only a very limited number of CASE tools support metamodel-based DSLs in such a generic way.

Overall, it seems to be common knowledge to choose a metamodel-based technique for defining a DSL, if

- the domain is well defined, and has a unique well accepted set of concepts,
- a model realized under the domain is not subject to be transferred into other domains.
- there is no need to combine your domain with other domains;

and to choose a technique based on UML profiles, if

- the domain may be combined with other domains, in an unpredictable way,
- models defined under the domain may be interchanged with other domains.

2.2 An Example DSL: SecureUML

In this paper, we use SecureUML [2] (and its dialects) as a running example of a family of DSLs. SecureUML is a security modeling language based on role-

based access control (RBAC) [13] with some generalizations. The abstract syntax of SecureUML is defined by a metamodel (see Figure 1). SecureUML supports

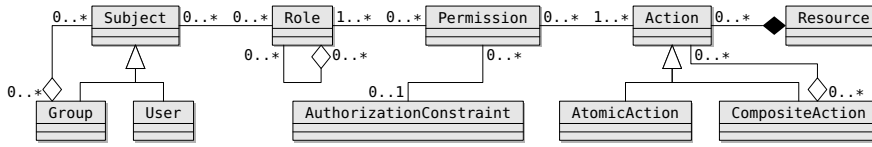

Figure 1. SecureUML Metamodel

notions of users, roles and permissions, as well as assignments between them: Users can be assigned to roles, and roles are assigned to specific permission. Users acquire permissions through the roles they are assigned to. Moreover, users are organized into a hierarchy of groups, and roles are organized into a role hierarchy. In addition to this RBAC model, permissions can be restricted by *Authorization Constraints*, which are conditions that have to be true (at run-time) to allow access.

Permissions specify which *Role* may perform which *Action* on which *Resource*. SecureUML is generic in that it does not specify the type of actions and resources itself. Instead, these are assumed to be defined in the *design modeling language* which is then "plugged" into SecureUML by specifying (in a SecureUML *dialect*) exactly which elements of the design modeling language are protected resources and what actions are available on them. Furthermore, a dialect specifies a *default policy*, i. e., whether access for a particular action is allowed or denied in the case that *no* permission is specified.

Currently, SecureUML supports two dialects: One for a component-based design modeling language, and one for a state-machine based modeling language. For example, a SecureUML dialect definition for UML class diagrams in the spirit of the ComponentUML dialect specifies classes, attributes and operations to be resources. The dialect also specifies, among others, the actions *create*, *read*, *update*, and *delete* on classes, *read* and *update* on attributes, and *execute* on operations.

SecureUML features a notation that is based on UML class diagrams, using a UML profile consisting of custom stereotypes. Users, Groups and Roles are represented by classes with stereotypes ≪secureuml.user≫, ≪secureuml.group≫, and ≪secureuml.role≫. Assignments between them are represented by ordinary UML associations, whereas the role hierarchy is represented by a generalization relationship. Permissions are represented as association classes with stereotype ≪secureuml.permission≫ connecting the role and a *permission anchor*. The attributes of the association class specify which action (the attribute's type) on which resource (the attribute's name) is permitted by this permission. Authorization constraints are (OCL) constraints attached to the association class. Attributes or operations on roles as well as operations on permission have no semantics in SecureUML and are therefore not allowed in the UML notation.

Figure 2 shows a part of a UML model of a group calendar applications together with an exemplary access control policy. The left part of Figure 2 shows

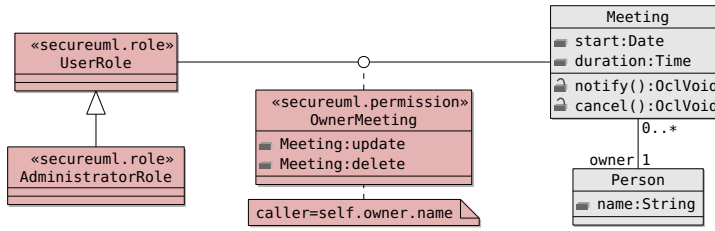

Figure 2. Access Control Policy for Class Meeting in Concrete UML Syntax

the access control policy for the class Meeting, whereas the right part shows the design model of the application. The design model consists of *Meetings* and *Persons*. The association class specifies a simple access control policy: only owners of meetings may delete them, or change the meeting data.

The association class (*OwnerMeeting*) has two attributes with type *update* resp. *delete*. This specifies that the associated role (*UserRole*) has the permission to update and to delete meeting objects. According to the policy, however, only *owners* of meetings should be able to do so. The property of being an owner of a meeting cannot be easily specified using a pure RBAC model. It is therefore specified using the authorization constraint $\mathsf{caller} = \mathsf{self.owner.name}$. For this purpose, we introduced a new keyword caller of type String into the OCL language that refers to the name of the authenticated user making the current call. Attaching this authorization constraints to the permission thus restricts the permission to system states where the name of the owner of the meeting matches the name of the user making the request.

3 Details of the Integration

MOF, even in the restricted form of EMOF, is a quite substantial language. For the purposes of presentation this paper, we restrict ourselves to the most-frequently used subset of MOF. We consider a metamodel as consisting of (meta-)classes with attributes and operations, as well as associations and generalization relationships between classes. Association ends can be further specified by multiplicities and whether they are ordered or not. Figure 3 shows this simplified meta-modeling language.

We can give a simple semantics to such metamodels by observing that such a metamodel directly corresponds to a first-order signature $\Sigma = (\mathcal{S}, <, \mathcal{F}, \mathcal{P})$, where $\mathcal{F}$ is a set of function symbols, and $\mathcal{P}$ is a set of predicate symbols: attributes and operations give function (symbols), and (n-ary) associations give (n-ary) predicate symbols. Instead of introducing a predicate *InstanceOfX* for each meta-class X, we use a many-sorted first-order signature, i. e., $\mathcal{S}$ is a set of

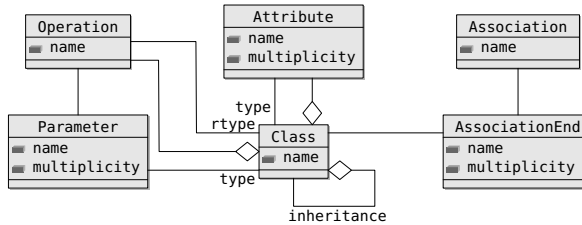

Figure 3. Simplified meta-modeling language

sorts, and for dealing with inheritance relations, we let this set of sorts be ordered by $\_ < \_$. An *instance* of a metamodel then corresponds to an algebra of the corresponding signature. To illustrate this, consider the SecureUML metamodel given above in Figure 1; this SecureUML metamodel corresponds to the following order-sorted signature: the set of sorts is

$$\mathcal{S} = \left\{ \begin{array}{l} \text{User}, \text{Group}, \text{Subject}, \text{Role}, \text{Permission}, \text{AuthorizationConstraint}, \\ \text{Action}, \text{AtomicAction}, \text{CompositeAction}, \text{Resource} \end{array} \right\},$$

where the sorts are ordered as follows: Group $<$ Subject, User $<$ Subject, AtomicAction $<$ Action and CompositeAction $<$ Action. Because there are no attributes or operations in the metamodel, the set of function symbols is empty, i. e.: $\mathcal{F} = \emptyset$. The set of predicate symbols is

$$\mathcal{P} = \left\{ \begin{array}{l} \text{GroupSubject}, \text{SubjectRole}, \text{RoleRole}, \text{RolePermission}, \\ \text{PermissionConstraint}, \text{PermissionAction}, \text{ActionCompositeAction}, \\ \text{ActionResource} \end{array} \right\}.$$

In a SecureUML dialect, this signature is then extended by additional sorts, which are ordered below the sort Resource.

A UML profile, in turn, consists of a set of stereotypes as well as a set of tagged values, which can be attached to certain UML modelelements. Stereotypes are essentially just strings, tagged values are (name, value) pairs, where the value can be, according to the standard, a typed UML model element, but in practice often also is just a string. For our purposes, we assume it is a string. Typically, the stereotypes in a UML profile are associated with constraints, which define the semantics of the modelelements tagged with this stereotype. In our approach, this is unnecessary, as the semantics is defined in terms of the metamodel, not the UML profile. However, it may be useful to define syntactic restrictions on the use of these stereotypes as OCL constraint. This is particularly useful in the case that the CASE tool can evaluate these constraint and in this way give feedback to the user.

There are basically three different ways for defining an integration between the DSL's metamodel and the UML profile for it:

1. describe the notation for metamodel elements, e. g., by defining for each metamodel element how it is represented in the UML notation.

2. describe the meaning of UML profile elements, e. g., by defining for each UML profile element which metamodel element it represents.
3. (externally) define a mapping from UML models using the profile to instances of the DSL metamodel.

Each of these approaches has their pros and cons. The third possibility is obviously the most flexible one, but has the disadvantage of being a bit heavyweight. The first two possibilities are conceptually very similar, they mostly differ in the point of view. As, in our view, the metamodel is the central definition of the language, it makes sense to put the notation definition in the metamodel, and not vice versa. For this reasons, we advocate the first possibility. The basic idea is to annotate each metamodel class with directly corresponding UML profile elements, i. e., stereotypes and tagged values. Additionally, OCL formulae may be used to calculate the associations and other relationships between the metamodel classes. This does not support arbitrary notations, but we argue it is still flexible enough to support sensible notations for most DSLs. In more detail, we define a UML profile for a metamodel as follows:

Metaclasses: we annotate each metaclass with two types of information: a UML modelelement type and a stereotype name.
The intended meaning is that every such UML modelelement with the given stereotype results in an instance of this metaclass. We do not require a metaclass to be represented by a UML class. This can considerably simplify the notation.

Attributes of metaclasses: we annotate each attribute of a metaclass with an OCL expression (to be evaluated over the UML model) in the context of the UML modelelement type that is used in the notation of the (meta-)class this attribute belongs to. This OCL expression is used to calculate the value of this attribute in an instance of the metamodel.

Binary Associations: We annotate each association end ae with an OCL expression ae, similar to the case of attributes. This expression is used to calculate, parametrized by `self`, the set of objects ae(`self`) associated to `self`. However, one now has to be careful: one has to ensure that opposite association ends give consistent results, i.e, if ae_1 and ae_2 are opposite association ends and $x \in \mathrm{ae}_1(y)$, then also $y \in \mathrm{ae}_2(x)$. In our experience, this was rarely a problem. For cases where this is a problem, or for cases where the association end value is not that easily calculated, we support another approach. If no OCL expression is defined for an association end, it's value is calculated using the inverse of the OCL expression of the opposite association end. We try to avoid this, however, as it may incur substantial overhead to calculate it this way.

n-ary Associations: The approach used for binary associations is not directly transferable to the general case of n-ary associations, as it would lose some of the information in the association. For this reason, we here annotate the association itself with an OCL expression that calculates the value of the associations, using an appropriate OCL TupleType as the result type.

Operations: There is no need to define a notation for operations. The behavioral specification of operations can be defined by pre- and postconditions in the metamodel of the DSL, as usual.

Inheritance: Because every instance of a subclass is automatically an instance of the superclass, there is no need to define a notation for inheritance relationships.

It is easy to see that given a metamodel for a DSL, one can always define a UML profile, in the way described above, that can be used to represent arbitrary instances of the metamodel: if nothing else, one can simply represent each DSL metaclass X by a UML class with stereotype «X». Associations between DSL metaclasses can then be represented straightforwardly by associations between the corresponding UML classes. The focus then lies on defining a UML profile that makes working with the DSL convenient. This requires some creativity on behalf of the DSL designer.

Sometimes, the notation defined in such a way poses difficulties which cannot be overcome. For example, consider the frequent case where we want to represent a particular metaclass and a particular associated metaclass by a UML classifier with its attributes and appropriate stereotypes. Using our approach so far would mean that one has to put a certain stereotype on each attribute of the UML classifier. As this is lot of work, we support a simple shortcut: instead of a stereotype, one can define an *anchor class* and an *anchor path*. The anchor class is a directly associated metamodel class. The anchor path is an OCL expression. Whenever this OCL expression evaluates to a UML model element that is mapped to an instance of the anchor class, this modelelement gets mapped also.

Although this definition of a notation maps DSL metamodel elements to UML profile elements, it actually defines a mapping from UML (profile) models to instances of the DSL metamodel. In certain, in our experience quite frequent, cases however, we actually can derive a mapping in both directions. Obviously, we can simply generate for each instance of a metaclass a corresponding UML modelelement with the appropriate stereotype. We then have to ensure, however, that the mapping back to the metamodel results in the same instance. For this, we have to ensure that the associations and attribute values "fit." If the OCL expressions for attribute and association end values are simple enough, for example only "property calls," we can do this by setting the property to the right value. This is the case for ComponentUML, for example, where the associations between the different resource types (entities, attributes, operations) are directly given by corresponding associations in the UML metamodel. Therefore these association end values are calculated by property calls like getOwner, getFeature, getParticipant, etc.

With these definitions so far, there are still some questions, for which we can currently only give partial answers:

- Defining a mapping between models (here, from a UML model to an instance of the DSL metamodel) always poses the question of how to ensure that the result of this mapping is well-formed. This means, the result should conform to the DSL metamodel.

For certain structural properties, like which associations between modelelements are possible, this is essentially done by a simple type-checking of the OCL expressions that calculate the association ends. For more complex properties, like multiplicities of association ends, or even OCL well-formedness constraints, this is, unfortunately, not possible.

- UML allows modelelements to be annotated with multiple stereotypes. This poses the problem of what to do, when a UML modelelement could be mapped to two different DSL modelelements, because it is annotated with two different stereotypes. There are essentially two strategies for handling these cases: (1) Simply map to both DSL modelelements, or (2) disallow these situations, and raise an error. Both strategies may be sensible in certain situations. A third strategy, namely mapping to only one of the DSL modelelements, and ignoring the others, is probably not useful.
- Given such a notation for a DSL, it may be that always mapping the complete UML model to an instance of the DSL metamodel is too expensive. This can happen with very large UML models, where additionally one wants to have a high degree of interactivity, for examples some continuous display, calculated on the basis of the DSL model. A solution for this would be, if we could not only map UML modelelements, but also UML model operations, like adding, deleting modelelements to corresponding operations on the DSL model. Also, editing the DSL model is sometimes more convenient directly on the DSL level (maybe through some specialized user interface, cf. Section 4). This would require, however, the converse ability: mapping DSL model operations to corresponding UML model operation. Unfortunately, both are not easy.

4 The Extension Architecture

For showing the applicability of our approach, we developed a plugin for the UML CASE tool ArgoUML (`http://argouml.tigris.org`) supporting arbitrary SecureUML dialects. Thus, our plugin does not only support one DSL, but a "family" of related DSLs. Overall, the plugin has three main purposes:

1. provide a DSL specific extension for storing models using the metamodel of the DSL and dialects thereof,
2. provide a synchronization facility to the model repository of the CASE tool using the UML notation as concrete syntax for modeling, and
3. provide a specialized concrete syntax in form of a special GUI.

Whereas the first two purposes will be present in any extension, providing a specialized GUI, i. e., a domain-specific user interface for the CASE tool, is optional.

We propose the architecture depicted in Figure 4 for developing such DSL-specific CASE tool extensions. Usually, CASE tools do not support domain-specific extensions based on metamodels directly. The fundamental idea of our architecture for supporting a metamodel-based DSL is to provide a separate MOF-compliant model repository (on the right side of Figure 4). We use this model-repository to store models specified using the DSL, e. g., SecureUML. Moreover, we allow for customizing this repository by loading dialects (also specified by

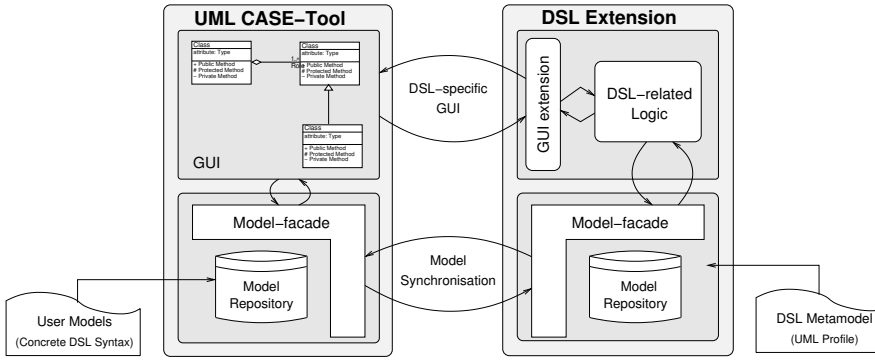

Figure 4. Generic DSL Extension Architecture

a metamodel) of our DSL. Recall that our DSL is defined by a metamodel that also encodes a UML profile. Based on this profile, the repository of the CASE tool (storing DSL models using the concrete, UML profile-based, syntax) can be synchronized with the model repository of our extension. The control-logic of the extension can now provide operations for modifying and analyzing specification that use the DSL, e. g., SecureUML.

Recall that our encoding provides a bi-directional mapping between the concrete UML syntax (which can be stored in the model repository of the CASE tool) and the abstract syntax of the DSL (which is stored in model repository of the DSL extension). Thus, we can store the complete model, including the domain-specific parts, in the model repository of the case tool and use its file format for storing models; i. e., there is no need for a DSL specific file format for storing models. Moreover, we can use the XMI export of the CASE tool for exporting standard UML models, containing the domain-specific parts as a UML profile. The XMI export of the model repository of the DSL extension can be used for exporting models in the abstract syntax of the DSL.

Overall, the CASE tool only has to provide a programming interface to its model repository, to be extensible with our approach. Thus, this approach can be used together with any CASE tool supporting a programming interface to its repository. On the one hand, by using a MOF-compliant repository for the DSL extension, even CASE tools using the Eclipse Modeling Framework (EMF) can be used with a MOF-based DSL definition. On the other hand, for CASE tools using a model repository supporting a concept of extents, e. g., as it is the case for ArgoUML using MDR, using a model extent is sufficient for our architecture.

Moreover, CASE tools that support extending their user interface can be further customized in a domain-specific way. For example, for our SecureUML extension of ArgoUML, we developed a SecureUML specific GUI as an alternative to the concrete UML syntax of SecureUML. Figure 5 shows an extension of ArgoUML's property pane for classes, for example. The SecureUML related properties in the middle of the pane allow for creating new roles and modifying the access control specification of the class Meeting (recall our example shown

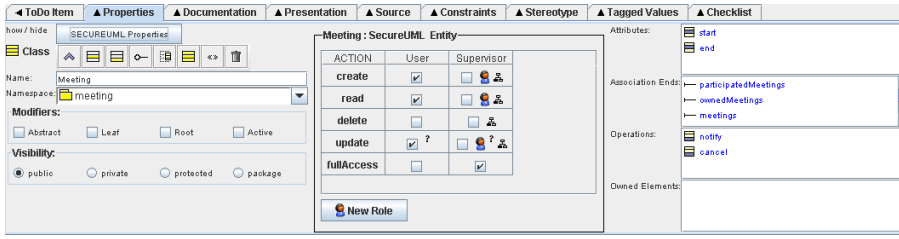

Figure 5. SecureUML Property Pane

in Figure 2). For example, in the column "User" we see that users are allowed to update objects of the class Meeting; the question mark denotes that this permission is restricted further by a constraint, i.e, self.caller=owner. Overall, such an extension of the GUI of the CASE tool can, in comparison to the concrete UML syntax of the DSL, provide a more concise presentation. Thus, this provides a first step into the direction of providing a domain-specific CASE tools, using a standard UML CASE tool as a kind of meta CASE tool, or framework.

5 Discussion

In this section, we discuss several alternatives and limitations of both our encoding and our extension architecture.

Repository synchronization. Our approach gears toward supporting the DSL by DSL-specific plugins or extensions that are integrated into a generic CASE tool. Given that the DSL has a well-defined metamodel, we assume that these extensions work on an API, i. e., a model-facade, that reflects this metamodel. In principle, there are two possibilities for implementing this model-facade:

1. the model facade can be implemented by a separate repository (or model extent, if supported) storing the complete DSL specific model information;
2. based on the representation using a UML profile, the DSL-specific model information can be mapped partially and on-demand into the DSL specific model repository.

In both cases, the DSL specific repository needs to be synchronized with the repository of the underlying CASE tool. The two approaches differ mainly in their synchronization strategy between the different repositories.

In the first case, we need to keep both repositories in sync at all times. Moreover, this synchronization needs to be bi-directional because we want editing capabilities in the DSL extensions. Providing a real-time, bi-directional synchronization of two model repositories is a technical challenge, e. g., one has to identify model elements after re-namings and one has to guarantee the consistency of both repositories at all times. In the second case, we only need to map the current scope, i. e., the parts of the model on which a domain-specific operation is performed, into a fresh repository. This solution does not suffer from the synchronization problem.

Based on this considerations, our prototype uses the on-demand mapping strategy (i. e., the second option), which has proven to be successful for our example CASE tool extension.

Query languages for metamodels. At first sight, our approach for specifying metamodel-based UML notations for DSLs heavily depends on a query language like OCL. Taking a closer look reveals that only a very limited subset of OCL (see Table 2) is used. Our approach only requires a query-language that comprises,

varDeclList	::=	[*varDeclList* `,`] *varDecl*
varDecl	::=	*simpleName* [`:` *type*] [`=` *expr*]
type	::=	*pathName* | *collKind* `(` *type* `)`
expr	::=	*literalExp* | *expr*`->`*simpleName*
		| *expr*`(`*expr*[`:`*type*][`=`*expr*]`,`*varDecl*`|`*expr*`)` | *expr*`(`[{*expr*`,`}*expr*]`)`
		| *expr*`(`*varDecl* `|` *expr*`)` | *expr*`->iterate(`*varDecl*[`;`*varDecl*]`|`*expr*`)`
		| *prefixOperator expr* | *expr infixOperator expr*
		| `if` *expr* `then` *expr* `else` *expr* `endif`
		| `let` *varDeclList* `in` *expr*
infixOperator	::=	`<` | `>` | `<=` | `>=` | `=` | `<>` | `and` | `or` | `xor` | `implies`
prefixOperator	::=	`-` | `not`
literalExp	::=	*collLiteralExp* | *primLiteralExp*
collLiteralExp	::=	*collKind*{{*collLiteralPart*`,`}*collLiteralPart*}
collKind	::=	`Set` | `Bag` | `Sequence` | `Collection` | `OrderedSet`
collLiteralPart	::=	*expr* | *expr*`..`*expr*
primLiteralExp	::=	`Boolean` | `Integer` | `Real` | `String`
pathName	::=	[*pathName*`::`]*simpleName*
simpleName	::=	*SIMPLE_NAME*

Table 2. OCL subset used for describing Metamodel-based UML Notations

informally, the following features: navigation over the metamodel, i. e., path expression, basic set operations (such as union and intersection), and a two-valued Boolean logic with (in-)equalities. These requirements are met by nearly all object-oriented programming or specification languages. Implementations, i. e., metamodeling frameworks often provide means for executing queries on a given (meta-) model. For example, EMF provides a query framework that supports many different languages. Whereas OCL is one of the supported languages, any language supported by the eclipse query framework would be sufficient.

Standards for metamodeling languages do usually not include a query language. Nevertheless, we propose to extend languages for metamodeling, e.g, KM3, with a standardized query language. This would enable the easy exchange of metamodels for DSL together with a concrete syntax (as UML profile) for the DSL. This guarantees the same mapping between metamodel and UML profile for all repository implementations supporting the specific metamodeling language.

In case of MOF, using OCL seems to be a natural choice: OCL is used widely in the UML and MOF standards and is more than sufficient for our needs. As an alternative, we could have also used pOCL [11], a variant of OCL that includes language constructs like while loops allowing turning OCL into a procedural programming language. Whereas our approach does not need the extensions pOCL offers, we can image that they could be useful, e. g., for computing the transitive closure of an association.

Providing a strong link between concrete syntax, abstract syntax, and semantics. Usually, the semantics of a DSL is defined with respect to the abstract syntax, e. g., by mapping the metamodel to a well-known semantic domain. In contrast, models are build using a concrete syntax which often provides a wealth of syntactic sugar. Therefore it is not obvious, that a modeler, using the concrete syntax, is conscious of the meaning of his models. Assuming a query language with a formal semantics, our approach provides a formal description of a bidirectional mapping from concrete syntax to abstract syntax. Thus, a formal semantics given in terms of the abstract syntax can be easily described in terms of the concrete syntax. This is especially important in situations where the mapping from concrete syntax to abstract syntax is not obvious; for more details, see [1] for example.

Supporting families of DSLs. Many DSLs, even though they are domain specific, are applicable in different contexts. For example, a notion of access control as provided by SecureUML, can be used within data models (e. g., class models) and behavioral models (state models). Whereas the underlying concept, e. g., role-based access control remains the same in both applications, model elements (and thus the subjects of SecureUML) that are restricted differ. We call this situation, where different dialects of the "same" DSL are needed, a *family* of DSLs. As all DSLs dialect of a family share common concepts, these concepts should be described in a reusable manner. Moreover, CASE tools extension can be implemented in a generic way, i. e., such a plugin supports a whole family of DSLs: support for a concrete dialect can be activated at runtime by loading the corresponding metamodel. For example, our prototypical ArgoUML plugin already supports arbitrary SecureUML dialects like ComponentUML or ControllerUML.

Supporting combinations of DSLs. Combining DSLs includes both the use of different members of the same DSL family within the same model and the simultaneous use of different DSL families. Both applications require the formal description of dependencies and incompatibilities between different DSLs. In principle, it should be possible to describe (e. g., using OCL) these dependencies as well-formedness requirements on the model. These well-formedness rules can be described in terms of invariants on the metamodel of the DSL and need to be checked before a fragment of a DSL is used. Moreover, every DSL and dialect thereof should be defined in its own (sub-) namespace; among others, this avoid ambiguities of the defined stereotypes.

6 Related Work

There are several DSLs that are defined using metamodels or UML profiles. Moreover, several DSLs, e. g., SysML [12], EPAL [6], or WebML [9] provide both a metamodel and a UML profile; however, all these DSLs define the metamodel and the UML profile independently from each other. Thus, none of these works can guarantee, that the metamodel and the UML profile define the same language. Overall, the majority of UML-related DSLs seems to be defined by profiles. We assume that the reason for this is the lack of UML CASE tools supporting metamodel-based DSLs.

Meta-CASE tools, that are not based on UML, like GME [3] or MetaEdit+ (`http://www.metacase.com`) allow the definition of a concrete representation for each metaclass in the abstract syntax. Within a concrete representation of a model, each instance of a metaclass is represented by its (visual) representation. A generalization of this approach, using OCL for selecting a refined definition of the concrete representation, is presented in [4]. The use of OCL allows, among other things, for choosing modifying the visual representation based on its context or usage of a specific model element. Nevertheless, all these approaches have in common that They focus on the presentation of the concrete syntax whereas we define a concrete syntax in terms of UML, i. e., independently from its concrete visual representation.

The work in [5] is quite similar to our approach: they use, externally specified, ATL [8] programs for mapping the abstract syntax to a concrete UML representation. In contrast, we encode this mapping, using OCL, within the metamodel of the DSL and, moreover, we aim for a bidirectional mapping.

There are several alternatives to MOF [10], e. g., EMF (`http://www.eclipse.org/emf`) and KM3 [7]. Our approach can be directly used for EMF-based metamodels using the EMF Query Framework which, among other query languages, supports OCL. In the case of KM3, the lack of a standardized query language prevents the direct transfer of our approach. Extending KM3 with a simple query language should be easy, though.

7 Conclusion

The debate if a UML-related DSLs should be defined using metamodels or profiles has a long history. Usually, DSLs are either defined using a metamodel-based approach or as a UML profile. Our work here describes one way for combining these approaches.

We presented an approach combining the advantages of both metamodel-based and UML profile-based approaches for defining DSLs. We achieve this by encoding the concrete UML syntax, i. e., a UML profile, into the metamodel of our DSL. Using such an encoding, one can describe syntax and semantics of a DSL in one place, providing a strong link between concrete syntax, abstract syntax, and semantics of the DSL.

Whereas our approach is not *directly* supported by many UML CASE tools, it can be implemented with only a very restricted interface to the model repository

of the CASE tool. As most of todays UML CASE tools are using standard model repositories, like EMF or MDR (`http://mdr.netbeans.org/`), such programming interfaces are provided and well-documented.

Moreover, our approach allows for developing plugins for any UML CASE tools providing a programming API for their internal model repository. For example, such a plugin could use internally a MOF-based storage for the metamodel-based DSL representation and communicate with the model repository of the CASE tool by using the API. Thus, plugins can be developed independently from the language used for describing the metamodel of the CASE tool.

Acknowledgment

We thank Marcel Beer for valuable discussions on the subject of developing a generic GUI for SecureUML and the work he did during his diploma thesis.

References

[1] D. Basin, M. Clavel, J. Doser, and M. Egea. A metamodel-based approach for analyzing security-design models. In MODELS *2007*, volume 4735 of LNCS, Heidelberg, 2007. Springer.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. ACM *Transactions on Software Engineering and Methodology*, 15(1):39–91, January 2006.

[3] M. J. Emerson, J. Sztipanovits, and T. Bapty. A MOF-based metamodeling environment. *Journal of Universal Computer Science*, 10(10):1357–1382, 2004.

[4] F. Fondement and T. Baar. Making metamodels aware of concrete syntax. In ECMDA-FA *'05*, volume 3748 of LNCS, pages 190–204, Heidelberg, 2005. Springer.

[5] B. Graaf and A. van Deursen. Visualisation of domain-specific modelling languages using UML. In ECBS *'07*, pages 586–595, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Enterprise privacy authorization language (EPAL 1.2), Nov. 2003. Available as `http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/`.

[7] F. Jouault and J. Bézivin. KM3: a DSL for metamodel specification. In FMOODS *'06*, volume 4037 of LNCS, pages 171–185, Heidelberg, 2006. Springer.

[8] F. Jouault and I. Kurtev. Transforming models with ATL. In MODELS *Satellite Events*, volume 3844 of LNCS, pages 128–138. Springer, 2005.

[9] N. Moreno, P. Fraternalli, and A. Vallecillo. A UML 2.0 profile for WebML modeling. In ICWE *'06 Workshops*, New York, NY, USA, 2006. ACM Press.

[10] Meta object facility (MOF) specification, 2005. OMG document formal/05-05-05. Also available as ISO/IEC 19502.

[11] MOF QVT final adopted specification, Nov. 2005. OMG document ptc/05-11-01.

[12] SysML specification v. 1.0, May 2006. OMG document ptc/06-05-04.

[13] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.