A Comparison of Standard Compliant Ways to Define Domain Specific Languages

Ingo Weisemöller and Andy Schürr

Real-Time Systems Lab
Technische Universität Darmstadt
D-64283 Darmstadt, Germany
{weisemoeller|schuerr}@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de/

Abstract. Domain specific languages are of increasing importance for today's software development processes. Their area of application ranges from process modeling over architecture description and system design to behavioral specification and simulation. There are numerous approaches for the definition and implementation of DSLs. Among others, the OMG offers UML profiles as a lightweight extension of a predefined multi-purpose language and MOF as a metamodeling language, which can be used to define DSLs from scratch. This contribution investigates various approaches to define DSLs, focusing on architectural description languages as an example. Besides the usage of UML profiles and the definition of an entirely new language with MOF, the adaption of the UML based on a metamodel extension is also considered. As a consequence of the shortcomings depicted for the different approaches, we suggest to combine UML profiles and metamodeling in order to compensate their weaknesses and take advantage of their benefits.

1   Introduction

Nowadays the usage of domain specific languages (DSLs) is of growing importance in software development processes. Languages like BPMN [13], ACME [5] or MATLAB/Simulink [19] offer support for various phases of the software development process. Most of them are built up from scratch, often by means of a proprietary metamodeling language. As a matter of fact this results in high efforts, when building tools based on these languages. The Object Management Group (OMG) has, therefore, introduced profiles as a mechanism to describe lightweight extensions of the Unified Modeling Language [16, 17] (UML) as well as the Meta Object Facility [14] (MOF) as a meta modeling language to provide standard methods for the definition of domain specific languages.

From the coexistence of different standards for the definition of DSLs arises the question for which languages UML profiles are appropriate and in which cases we need to define a heavyweight extension or specify a new metamodel. On the one hand, UML is wide spread and well known, and commercial tool support is available at least for editing UML diagrams. Constraints on the models can

be defined in the Object Constraint Language [15] (OCL), for which support is currently available in some research projects [3] and in a few commercial tools. On the other hand, customized languages do not only offer potentially greater expressive power than profiles, and allow the usage of domain specific modeling elements. Their users also benefit from the availability of code generators, which results in lower efforts to build analysis tools and editors with a customized concrete syntax. Besides that, a customized language may be smaller and easier to learn than is UML.

The following sections deal with various approaches to define a DSL for software architectures and architecture families. As we primarily address users of UML, we focus on profiles and MOF as metamodeling techniques. Indeed we accept to run the risk of omitting the advantages of other languages, but this makes it much easier to bring together the benefits of metamodeling and profiles, since we already have MOF-QVT [11] as a standard for model-to-model transformations, which we want to use to combine profiles and metamodeling in the future. Altogether, we distinguish between three approaches of metamodeling:

– The description of a lightweight extension of the UML by using profiles and the equivalent extension of the UML metamodel. (Section 3)
– Using inheritance to extend the UML metamodel fragment that deals with components, thus introducing subclasses of the metaclasses defined in the UML 2.1 specification. (Section 4)
– The specification of a metamodel for MVC architectures in MOF. (Section 5)

For each approach we will consider the models and the metamodel of both the MVC design pattern and the architecture of the "Java Pet Store" (cf. section 2). We will then survey each approach with respect to the clarity of the corresponding metamodels and their semantics, usability for modelers and metamodelers, ease of defining constraints, and tool support.

2 Running Example

Each approach to define DSLs will be discussed based on the example of a language for software architectures. In particular, we will focus on the formal description of architectural guidelines and how to provide tool support for the automated checking of these guidelines. As an example for a concrete software architecture the Java Pet Store 1.1 web application, Sun's sample application for J2EE technology, will be used. It has been designed according to the Model View Controller (MVC) design pattern, and a detailed textual description of the architectural concepts [20] is available. Thus we have sufficient information about its architectural concerns without being biased due to the usage of a certain modeling language. Figure 1 shows an excerpt from the architecture in a notation based on UML component diagrams.

The basic idea of the MVC design pattern is to decouple a system into three areas of responsibility, each of which is improved in terms of extensibility, maintainability and replacability. We want to ensure architectures preserve this separation of concerns by demanding that each component of a system may only
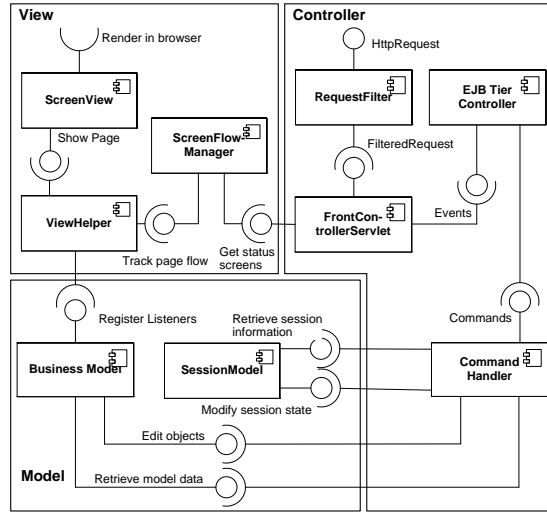
Fig. 1. Simplified architecture model of the Java Pet Store

provide interfaces of the same area of responsibility as the component. Additionally, we will distinguish between *critical*, *stable*, and *unstable* components, and define whether the usage of a component is *strict*, in which case the used component must be at least as stable as the one that uses it.

3 Using UML profiles

Profiles have been introduced by the OMG to allow users to adapt the UML to their personal needs. According to the UML Superstructure specification [17], profiles are not supposed to extend the UML metamodel. Thus they are conceptually interchangeable between tools, using XMI [12] as data format, though this results in technical difficulties in practice, because most commercial tools do not completely comply to the standard. However, although the specification states that the UML metamodel is not extended by profiles, it describes metamodels that are equivalent to a given profile. There has been a considerable amount of publications on the modeling of architectural styles with UML profiles before, most of which deal with UML 1.x (cf. section 6).

A sample profile for the distinction between model, view or controller elements is shown on the left side of figure 2. To keep the example small, the separation into the three areas of responsibility is limited to Interfaces and Components. The right side of the same figure shows an extension of the UML metamodel that is equivalent to the profile according to the UML Superstructure specification [17].

For the specification of the desired stability of a Component we introduce the stereotype Importance, which provides an attribute of the Stability enumeration
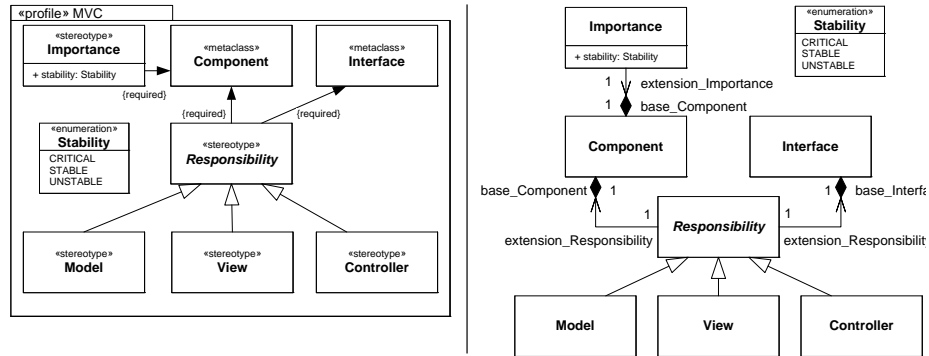
Fig. 2. A UML profile for MVC architectures and its MOF equivalent

type. An abstract stereotype Responsibility that generalizes the three stereotypes Model, View, and Controller has been defined for the decoupling of the system into areas of responsibility. This should ensure that only one of the stereotypes can be applied to each instance of the extended metaclasses. In fact this depends on the mapping of inheritance between stereotypes to the metamodel. It relies on the creation of an extension_Responsibility link each time a substereotype is applied. This is what commercial tools like Enterprise Architect in fact pretend to do, but the UML specification does not clarify how to handle inheritance of stereotypes. Therefore, the creation of associations for the concrete classes with properties base_Interface, base_Component, extension_model, extension_View and extension_Controller must also be taken into consideration. If these associations are marked as subsets of the one shown in the figure, the application of only one stereotype would be ensured for the above reason. If a tool behaves different in this point, this may force the user to introduce appropriate constraints and make model interchange by means of XMI difficult or even impossible.
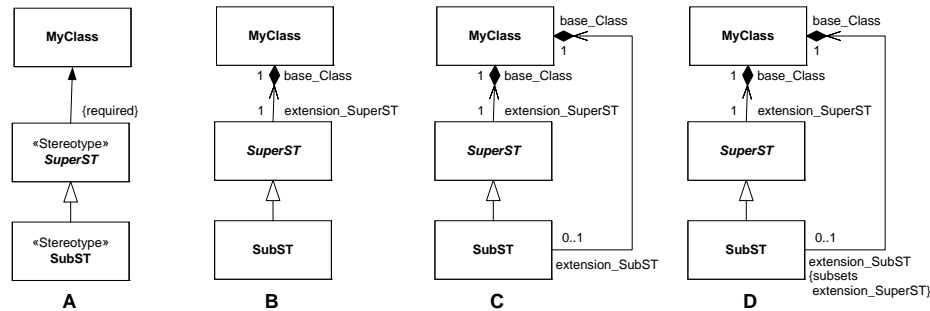

Fig. 3. Generalization of stereotypes (A) and mappings to a MOF model (B–D)

Figure 3 shows a small example of inheritance between an abstract and a concrete stereotype (A) and possible mappings to an equivalent MOF model (B–C). It must be pointed out that none of them can be applied by a tool directly, since this cannot modify its own metamodel at runtime. Instead, appropriate behavior must be ensured by other means, but the implementation is left up to the tool developers. Getting back to the figure, we see that Mapping B is the one applied in figure 2, introducing a single association in the MOF model. This ensures exactly one subtype of SuperST (only SubST is available in the example) is applied to each instance of MyClass. Mapping C introduces an additional association between each substereotype and the extended class, but this association does not subset the one between the superclass and the extended class. If the superclass is abstract and the stereotype is required (as in the example), there are three possible ways to reflect the application of SubST to a MyClass instance. The first one is to set only the extension_SuperST property, in which case the other association is useless. The second way is to set only the extension_SubST property, which results in a violation of the 1 multiplicity of the extension_SuperST association end, though we would expect this to result in a valid model. The third possibility is to create two links (one for each association), which will result in a violation of the 1 multiplicity of the extension_SuperST association end in case multiple substereotypes are applied to an element. The mapping shown in figure 3 D is only affected by the latter problem, since the application of an instance of SubST to an instance of MyClass results in a link in both associations due to the subsets property of the extension_SubST association end.

For the remainder of this section we will assume that the extension of classes by stereotypes is mapped to the metamodel as shown in figures 2 and 3 B. Based on this, the following constraint can be added to Responsibility to ensure that all Interfaces and their providing Components are of the same responsibility:

```
context Component inv: self.provided->forAll(i |
  i.extension_Responsibility.getMetaClass() =
  self.extension_Responsibility.getMetaClass())
```

As you will have noticed, the provided property and getMetaClass() method are not defined in our profile, but are available from the UML metamodel respectively the MOF reflection. Moreover we want to specify whether a Usage relation between Components is strict, i.e. whether it may point from a component of a certain level of stability to a less stable one. Therefore, we introduce a stereotype Strict, which extends the Usage metaclass, and attach this constraint to it:

```
context Usage inv: Stability.ownedLiteral->indexOf(
    self.client.extension_Importance.stability)
  >= Stability.ownedLiteral->indexOf(
    self.supplier.extension_Importance.stability)
```

Finally, let us look at the appliance of this profile to the concrete architecture (figure 4). We will restrict this to the EJB Tier Controller, since the effect on other components is basically the same. The concrete syntax for our extended
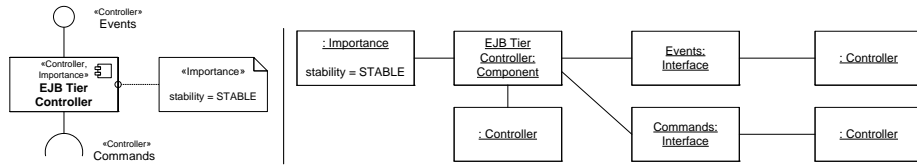
Fig. 4. The UML profile applied to the architecture and an equivalent object diagram

metamodel is defined by the UML specification. A minor issue is the presentation of the stability tag in a note, where an additional attribute for the EJB Tier Controller would be preferable.

Altogether, modeling of architectural styles based on UML profiles is quite an extensive approach. It requires an OCL expression for the rather simple constraint on relationships between components and interfaces. All constraints are rather complicated since we have to navigate between stereotypes and the extended classes, and there are some ambiguities concerning inheritance between stereotypes. The advantage of UML profiles, besides reusability of the UML metamodel, is the variety of existing tools we can use to describe architectures now. Support for UML profiles is provided by a series of commercial products, though not all of them do support UML 2 yet. These tools allow the definition of profiles as well as their appliance to a model, thus being usable as an editor not only for UML, but also for DSLs defined by means of UML profiles. Unfortunately, the validation of OCL constraints does not work properly in general.

4 Extending the UML metamodel

This section addresses heavyweight extensions of the UML by means of a MOF tool. This differs from the approach described in section 3, for which a UML tool supporting profiles is sufficient. Unfortunately, the definition of a metamodel *extension* is not that easy. If the creation, deletion and modification of arbitrary elements was allowed, it might be "extended" to any other metamodel. To our knowledge there is no generally accepted definition of a metamodel extension. However, the OMG suggests to use the package merge concept from the UML Infrastructure [16, pp 162ff] for this. Therefore, we limit extensions of a metamodel to the merge of an arbitrary package into the outermost package of the original metamodel.

Based on this, one obvious way to distinguish between interfaces provided by model, view and controller components is to introduce subclasses for Component and Interface from the UML metamodel. Unfortunately, we cannot make existing metaclasses abstract to ensure only the subclasses are instantiated. This is due to the definition of package merge: "For all matching classifier elements: if both matching elements are abstract, the resulting element is abstract, otherwise, the resulting element is non-abstract", which implies that a non-abstract UML metaclass remains non-abstract in our extended metamodel. Therefore, we have
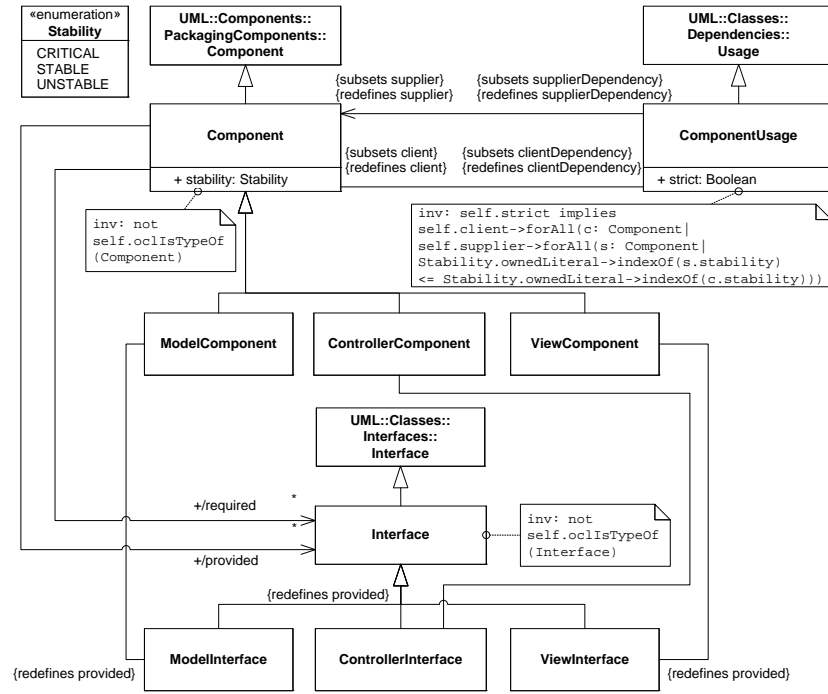
«enumeration»
Stability

CRITICAL
STABLE
UNSTABLE

**UML::Components::
PackagingComponents::
Component**

**UML::Classes::
Dependencies::
Usage**

Component

+ stability: Stability

{subsets supplier}
{redefines supplier}

{subsets supplierDependency}
{redefines supplierDependency}

ComponentUsage

+ strict: Boolean

{subsets client}
{redefines client}

{subsets clientDependency}
{redefines clientDependency}

```
inv: not
self.oclIsTypeOf
(Component)
```

```
inv: self.strict implies
self.client->forAll(c: Component|
self.supplier->forAll(s: Component|
Stability.ownedLiteral->indexOf(s.stability)
<= Stability.ownedLiteral->indexOf(c.stability)))
```

ModelComponent   **ControllerComponent**   **ViewComponent**

**UML::Classes::
Interfaces::
Interface**

+/required   *

Interface

```
inv: not
self.oclIsTypeOf
(Interface)
```

*

+/provided

{redefines provided}

{redefines provided}   **ModelInterface**   **ControllerInterface**   **ViewInterface**   {redefines provided}

Fig. 5. Excerpt from the UML metamodel extended by subclasses

to attach the constraints shown in figure 5 to Component and Interface to ensure
no instances of them can be created. The new subclasses do not necessarily have
additional attributes, but they allow us to redefine associations and introduce
constraints for our needs to describe MVC architectures.

As with UML profiles, we will try to express that an interface of a specific type
must be provided by a component of the same type. Therefore, we introduce an
association between each subclass of Component and the corresponding subclass
of Interface, which redefines the association between the superclasses as shown
in figure 5. We can also define several categories of stability in an enumeration
and add a new attribute of this type to a subclass of Component. Additionally,
we introduce a subclass ComponentUsage of the Usage relation from the UML
metamodel and add the attribute strict to it. The association ends between
this class and Component redefine those from the UML metamodel. Thus all
usages of components must be properly modeled by an instance of the new
ComponentUsage class. The following constraint ensures a component does not
depend on a less stable one if the strict attribute is set:

```
inv: self.strict implies self.client->forAll(c: Component|
  self.supplier->forAll(s: Component|
  Stability.ownedLiteral->indexOf(s.stability)
  <= Stability.ownedLiteral->indexOf(c.stability)))
```

Each component and interface in a model must now be represented by an instance of one of the new subclasses instead of Component and Interface from the original UML metamodel. The association refinements and the constraints make sure components provide only interfaces of appropriate types and do not strictly depend on less stable components. The concrete architecture will basically look the same as the one shown in Figure 1, except that a concrete syntax for the new subclasses and relation needs to be introduced.

The FrontController component from our sample architecture provides a view interface which offers frequently used pages, e.g. login or error screens. This violates the association refinements specified above, so a tool based on our extended metamodel would have prevented a software architect from creating this model. Instead, he might have introduced an additional view component, which gets status information from the FrontController and creates status screens from it.

As we see, the formulation and check of some basic constraints can be accomplished in an UML metamodel extended by inheritance. On the other hand, elements from the original metamodel which have become unnecessary are still present in the extended metamodel. The most serious problem is the loss of compatibility to existing UML tools. So, in order to apply this approach, one would have to modify an existing or write a new tool according to the new metamodel.

5 Defining a new metamodel

The last approach discussed in this contribution is the specification of domain specific languages from scratch by means of a metamodeling language. In contrast to an increasing number of proprietary metamodeling languages, the Meta Object Facility [14] (MOF) has been introduced by the OMG to describe models of metadata in a format independent of platform and manufacturer. A small number of tools supporting the specification of models using MOF [1] or its less expressive subset EMOF is available. In this section, the adequacy of MOF for the specification of DSLs will be discussed.

Figure 6 shows a simple metamodel to distinguish between model, view and controller components or interfaces. It is similar to the extended UML metamodel discussed in section 4, but there are some differences due to the fact that we did not start with a predefined metamodel: First of all, the subsystems of an architecture are no longer referred as components, but as Modules. When building a metamodel from scratch, arbitrary names can be given to the model elements, which allows the usage of identifiers specific to the users organisation for instance. Second, the superclasses Module and Interface can be abstract classes, so we do not have to define a constraint to ensure they cannot be instantiated. Third, the provided property of a Module is the derived union of its subsets (which may be determined by a different derivation rule in turn), instead of the derivation rule taken over from the UML into our extended metamodel.

Since we have not introduced any notation, an excerpt from the architecture of our sample application is shown as an object diagram in figure 7. To be able to build a graphical editor for DSLs, a concrete syntax needs to be defined;
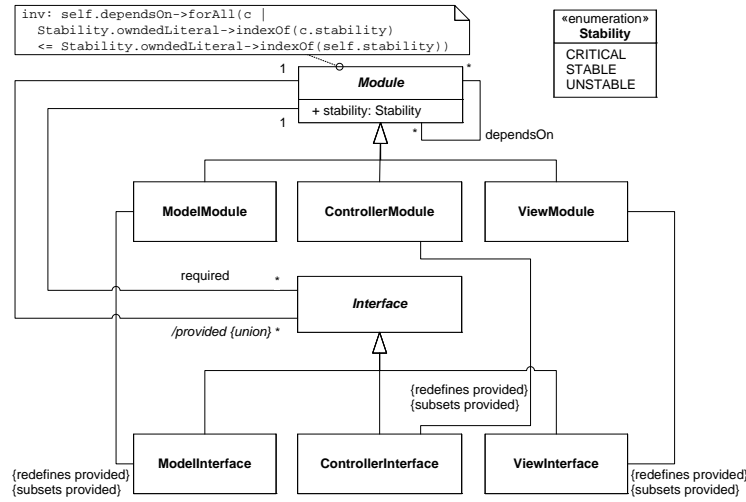
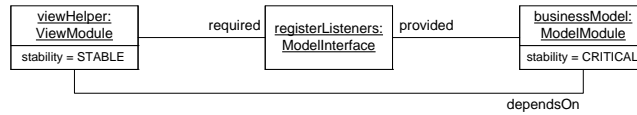**Fig. 6.** A MOF model for the MVC design pattern

Fig. 7. An excerpt from the architecture model of the Pet Store

frameworks for building editors and tools based on MOF or EMOF models are currently being developed by several organisations, e.g. [10]. The figure shows the coupling of the model and the view subsystem, which is realized by the reg- isterListeners interface. Due to the usage of this interface the viewHelper module depends on businessModel, which requires the businessModel to be at least as stable as the viewHelper. The corresponding constraint from the metamodel is fulfilled in the example, so this is a valid model.

As we have seen, the most important advantage of building a new metamodel are the extensive possibilities to adapt it to our needs and preferences. The association concepts of MOF provide a more precise and convenient way to specify relationships. Besides that, the elements can be defined in a way that makes the OCL constraints more compact than in UML profiles. Basic support for MOF based metamodeling is available as well as support for building tools on top of these metamodels. Nevertheless, building tools is still a complex and extensive task. Moreover, the introduction of a new language may require more practice than the extension of UML by means of profiles, which requires neither an entirely new concrete syntax nor new tools for the introduction of domain specific modeling elements.

6 Related Work

Several publications address the definition of DSLs, many of which deal with extensions of the UML or the specification of a new DSL by means of MOF.

Dong and Yang suggest to use UML profiles to describe architectural styles in [4], but they focus on the visualization of patterns in system design rather than consistency checking, though a few examples of constraints are given.

A publication of Henderson-Sellers and Gonzalez-Perez [7] investigates the differences between stereotypes in UML 1 and UML 2 and points out some issues about their specification from a set theoretical point of view. However, it contains some serious flaws. For instance, the authors state that "the Stereotype metaclass in UML 2.0 is a subtype of Class, so only classes can be stereotyped", where the first has nothing to do with the latter (and it should be mentioned that "classes" in this sense are more than classes in class diagrams), or they point out the "inability of stereotypes to express behavior", although this does obviously not hold for stereotypes which extend behavioral constructs.

The authors of [6] use a framework for model-to-model transformations to map domain specific languages to UML. As discussed in the previous sections of this contribution, they state that code generation and automated model analysis usually come along with the introduction of DSLs, whereas UML is primarily used as a target language for the visualization of models. However, their approach does not make use of UML profiles yet, but is focused on basic UML instead.

Besides these scientific projects there are a few commercial, proprietary meta-case-tools available [8], which in general lack interoperability, because they do not comply to a standard meta modeling language, and modularization concepts.

7 Conclusions

Our comparison of various approaches to define domain specific languages has shown the benefits and drawbacks of UML profiles and metamodeling. Profiles are supported by current CASE tools, but the concepts to refine associations are rather weak in comparison to those of a metamodeling language. They also suffer from a lack of flexibility, which makes the specification of constraints for consistency and integrity checking more complicated than necessary. In addition to these issues, there are some uncertainties about how to map profiles to an equivalent metamodel. Domain specific languages built up from scratch or as a heavyweight extension of the UML are better suited for these purposes, but they require high effort on tool building to be really usable for model editing.

8 Future Work

From the benefits and drawbacks of the approaches discussed in the previous sections follows the desire to combine the advantages of UML profiles and meta-modeling. More precisely, a way to define DSLs and make them usable without building entirely new tools is required. For this purpose, we suggest to define a

	UML profiles	UML extension	New Metamodel
Expressive power	-	+	+
Flexibility	-	o	+
Clarity of semantics	-	+	+
Simple constraints	-	o	+
Model Notation	-	-	+
Tool support	+	-	-

Table 1. Overview of approaches to specify DSLs

mapping from a limited set of domain specific languages to UML profiles and vice versa, which will make the use of commercial CASE tools as editors possible, but enable us to revert to the possibilities provided by metamodeling tools for integrity and consistency checking of the models. For the definition of a DSL and adaption of existing tools to this language we want to perform the following steps as shown in figure 8:
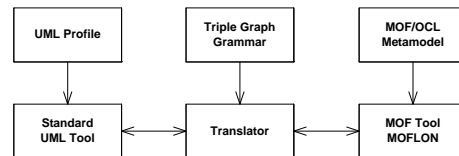

Fig. 8. Combination of UML profiles and metamodel based technologies

1. The abstract syntax of a DSL is defined in a MOF-compliant metamodeling tool like MOFLON [1]. OCL constraints may be used to define static semantics of models described in the DSL.
2. A UML Profile is used to define the concrete syntax of the new language with constructs similar or identical to those used by UML.
3. An implementation of QVT based on Triple Graph Grammars [9, 18] is used to translate the stereotyped UML model into an instance of the metamodel and vice versa.

The combination of UML profiles and metamodel based technologies is supposed to be a systematic replacement for extensive usage of profiles [2], reducing the effort of implementations to ensure the proper use of such profiles.

References

1. C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications:*

    Second European Conference, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

2. David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 100–109, New York, NY, USA, 2003. ACM Press.

3. D. Chiorean, B. Demuth, M. Gogolla, and J. Warmer. *OCL for (Meta-)Models in Multiple Application Domains*, volume 4364/2007 of *Lecture Notes in Computer Science*, pages 152–158. Springer Berlin/Heidelberg, 2007.

4. J. Dong and S. Yang. Visualizing design patterns with a uml profile, 2003.

5. David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

6. Bas Graaf and Arie van Deursen. Visualisation of Domain-Specific Modelling Languages Using UML. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 586–595, Washington, DC, USA, 2007. IEEE Computer Society.

7. Brian Henderson-Sellers and Cesar Gonzalez-Perez. Uses and abuses of the stereotype mechanism in uml 1.x and 2.0. In *MoDELS*, pages 16–26, 2006.

8. Hosein Isazadeh and David Alex Lamb. CASE Environments and MetaCASE Tools. Technical report, Queen's University School of Computing, 1997.

9. A. Königs. *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Technische Universität Darmstadt, 2007. To appear.

10. Marc Minas. Generating Visual Editors Based on Fujaba/MOFLON and DiaMeta. In *Proc. 4th Fujaba Days*, Technical Report tr-ri-06-275, pages 35–42. University Paderborn, 2006.

11. Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Nov 2005. http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf.

12. Object Management Group, Inc. MOF 2.0/XMI Mapping Specification, v2.1, Sep 2005. http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf.

13. Object Management Group, Inc. Business Process Modeling Notation Specification, Feb 2006. http://www.omg.org/cgi-bin/apps/doc?dtc/06-02-01.pdf.

14. Object Management Group, Inc. Meta Object Facility (MOF) Core Specification, Jan 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf.

15. Object Management Group, Inc. Object Constraint Language, May 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

16. Object Management Group, Inc. Unified Modeling Language: Infrastructure, Feb 2007. http://www.omg.org/cgi-bin/apps/doc?formal/07-02-06.pdf.

17. Object Management Group, Inc. Unified Modeling Language: Superstructure, Feb 2007. http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf.

18. A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

19. Simulink - simulation and model-based design, 1994–2007. http://www.mathworks.com/products/simulink/.

20. Sun Microsystems, Inc. Java pet store architectural overview, 2001. http://java.sun.com/blueprints/code/jps11/archoverview.html.