

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 (DSLs) 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 pre-defined 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 (ADLs) as an example. Besides the usage of UML profiles and the definition of a completely 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 [20], ACME [8] or MATLAB/Simulink [29] offer support for various phases of the software development process. Most of these languages 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 [23, 24] (UML) as well as the Meta Object Facility [21] (MOF) as meta modeling language to provide standard methods for the definition of domain specific languages. Another way to define a DSL is to take the UML metamodel as a basis and extend it according to the users' needs.

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. The advantages of using UML only are obvious: it is wide spread and well known, and commercial tool support is available at least for editing UML diagrams. Constraints on the models described in the DSL can be defined in the Object Constraint Language [22] (OCL), for which support is currently available in some tools provided by research institutions [5] and also in a few commercial tools [30]. In comparison to UML profiles, customized languages do not only offer potentially greater expressive power 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 using a customized concrete syntax. Besides that, if we do not need to model all aspects we can express using UML, a customized language may be smaller and easier to learn than is UML.

The following sections deal with these various approaches to define a metamodel for software architectures and architecture families. Because our approach primarily addresses 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 by this limitation, but it makes it much easier to bring together the benefits of metamodeling and profiles, since we already have MOF-QVT [18] as a standard for model-to-model transformations, which we want to use to combine profiles and metamodeling in the future. Each approach is investigated taking the Model View Controller (MVC) pattern and the sample application “Java Pet Store” described in section 2 as examples. Altogether we distinguish between three different approaches of metamodeling in the following:

- 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 dedicated metamodel for MVC architectures in MOF. (Section 5)

For each approach we will take a look at the models and the metamodels of both the MVC pattern and the concrete architecture of the Pet Store. We will then discuss the pros and cons of each approach in terms of clarity of the corresponding metamodels and their semantics, usability for modelers and metamodelers, ease of defining constraints in OCL, and tool support.

2 Running Example

For the remainder of this paper we will discuss the various approaches to define DSLs based on the example of a language for software architectures and architectural styles. The design of software architectures is part of the software development process in many process models used in current practice, e.g. waterfall model [26], spiral model [4] or rational unified process [25]. On the one

hand, there exist relationships between the description of the software architecture and documents generated during the other phases of the development process; for instance, the architecture must be consistent with the requirements specification and the actual source code must match the architecture. On the other hand, a formal way to describe a set of guidelines, best practices, and architectural styles as well as a way to apply them to the actual pieces of software developed by a specific company or organisation is required. We want to focus on how to describe such styles in a formal way and how to provide tool support for the automated checking of architectural guidelines in the following.

As an example for a concrete software architecture the Java Pet Store 1.1 web application, Sun's sample application for J2EE technology [33], will be used. The Java Pet Store has been designed according to the Model View Controller (MVC) design pattern [31], and a detailed textual description of the architectural concepts [32] as well as the source code is available. Thus we have sufficient information about its architectural concerns without being biased due to the usage of a certain formal architecture modeling language. Figure 1 shows an excerpt from the Pet Store architecture in a notation based on UML component diagrams.

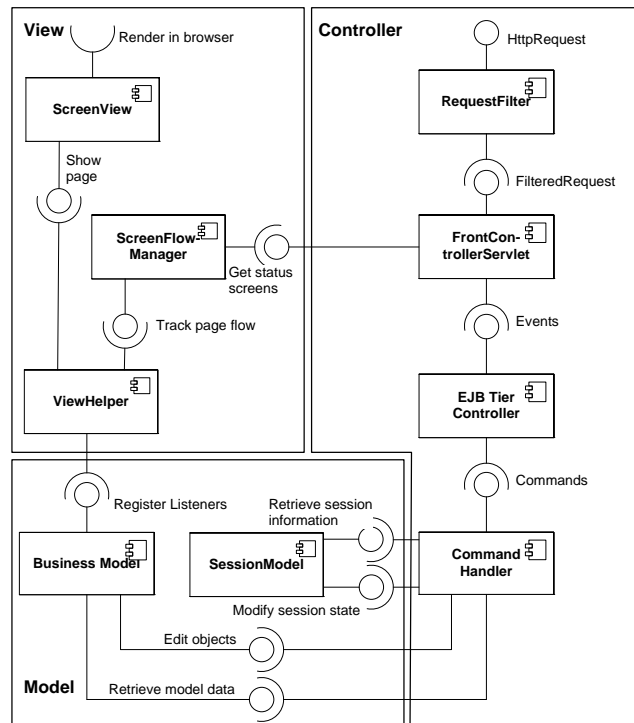


Fig. 1. Simplified architecture model of the Java Pet Store

The basic idea of the object oriented 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. In the following sections we want to ensure that MVC architectures preserve this separation of concerns by demanding that each component of a system may only provide interfaces which belong to the same area of responsibility as the component. Additionally, we will distinguish between the categories *critical*, *stable*, and *unstable* for the required stability of components, and to define whether the usage of a different 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

The UML profiles packages has been defined by the OMG to allow users to adapt the UML to their personal needs. According to the UML Superstructure specification [24], profiles are not supposed to extend the UML metamodel. Thus they are conceptually interchangeable between various UML tools using XMI [19] as a format for data and metadata exchange, though it needs to be mentioned that the XMI data exported and imported by commercial tools does not completely meet the requirements of 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 [11, 15]. Selonen et al. also describe an approach to validate architectures against guidelines using UML 1.4 profiles and OCL [28], although no examples of OCL constraints are given in this paper.

A sample profile for the distinction between model, view or controller components and interfaces 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 [24].

For the specification of the desired stability of a **Component** the stereotype **Importance** has been introduced. For the actual stability value it provides an attribute of the **Stability** enumeration type. An abstract stereotype **Responsibility** that generalizes the three stereotypes **Model**, **View**, and **Controller** has been defined for the decoupling of the system into the corresponding 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 assumption depends on how generalization of stereotypes is mapped to the UML metamodel when applying the corresponding profile. It relies on the creation of an **extension_Responsibility** link each time an instance of the concrete classes **Model**, **View** or **Controller** is created. This is what commercial tools like Enterprise Architect [30] in fact pretend to do, but the UML specification does not finally clarify how to handle

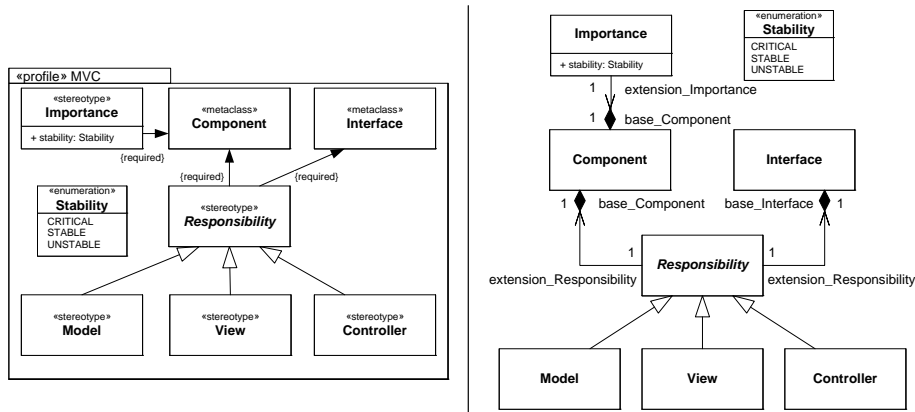


Fig. 2. A UML profile to distinguish between areas of responsibility and to describe stability requirements in MVC architectures and its MOF equivalent

inheritance of stereotypes. Therefore the possibility of creating associations for each of 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 reason described above. If other tools behave different in this point and introduce further properties, this may force the user to introduce appropriate constraints and make model interchange by means of XMI difficult or even impossible.

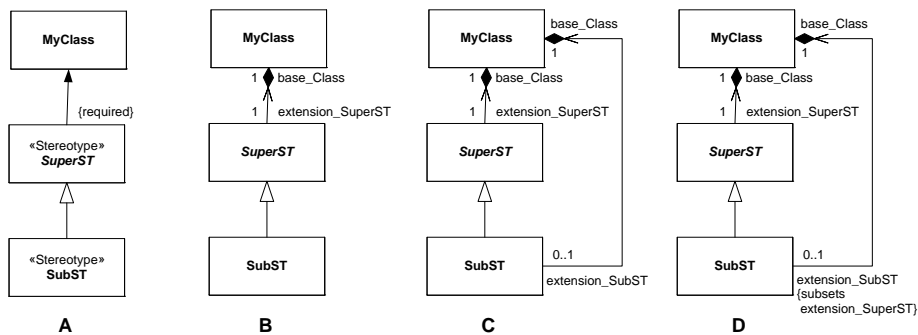


Fig. 3. Generalization of stereotypes (A) and possible mappings to an equivalent MOF model (B–D)

Figure 3 shows a small example of inheritance between an abstract and a concrete stereotype (A) and the possible mappings of this to an equivalent MOF model (B–C). Before discussing the various mappings it must be pointed out

that none of them can be applied by a UML tool directly, since the metamodel of the tool cannot be modified at runtime. Instead, it must be ensured by other means, that the models behave as implied by the mapping, but the choice how to implement this 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 makes sure exactly one of the subtypes 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 not affected by this 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 assumption, the following constraint can be added to `Responsibility` to ensure that all `Interfaces` and their providing `Components` are of the same responsibility stereotype:

```
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 explicitly in our profile, but are predefined in the UML metamodel respectively the MOF reflection. Moreover we want to be able 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. For this purpose, we can introduce a stereotype `Strict`, which extends the `Usage` metaclass, and attach the following constraint to it:

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

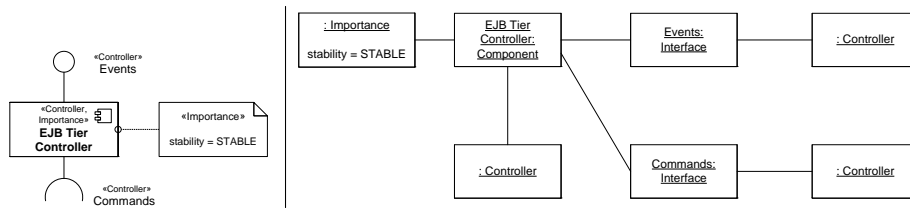


Fig. 4. The UML profile applied to the architecture and an equivalent object diagram

Finally, let us take a look at figure 4 now, which shows the appli-
 cation of this profile to the concrete architecture. We will restrict this to the EJB Tier Controller, since the effect of applying our profile on other components is basically the same. The concrete syntax for our extended metamodel is defined by the UML specification. A minor issue is the presentation of the **stability** tagged value in a note, where an additional attribute for the EJB Tier Controller would be preferable.

Taking a look at the complexity of the definition of our architectural style, we see that modeling based on UML profiles is quite an extensive approach. We had to introduce an OCL constraint for the rather simple relationship between the responsibility area of components and interfaces. The constraint for strict dependencies is rather complicated since we are forced to navigate between stereotypes and the extended classes, and there are ambiguities in the UML specification concerning inheritance between stereotypes. The primary advantage of UML profiles, besides reusability of the UML metamodel, is the variety of existing tools we can use to describe the concrete architectures. Support for UML profiles is, among others, provided by commercial products like Rational Rose or Enterprise Architect, though not all of these tools do support UML 2 yet. These tools allow the definition of profiles as well as their application to a concrete 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 the specification of heavyweight extensions of the UML metamodel by means of a MOF tool. This extension clearly differs from the lightweight extensions described in section 3, for which a UML tool supporting profiles is sufficient. Unfortunately, the distinction between a metamodel extension and the introduction of a new metamodel is not that easy. If the creation, deletion and modification of arbitrary elements from a metamodel was allowed, it might obviously 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 for metamodel extensions. We will reuse the transformation rules for package

metamodel remains non-abstract in our extended metamodel. Therefore, we have to attach the constraints shown in figure 5 to the metaclasses **Component** and **Interface** to ensure no instances of them can be created. Note that this does not have an impact on the original classes in the **Components** and **Dependencies** package, which we cannot prevent from being instantiated. The introduced subclasses do not necessarily have new defaults or 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 always be provided by a component of the same type. Since we can add new associations to the metamodel, we can ensure this by introducing an association between each subclass of **Component** and the corresponding subclass of **Interface** and making it redefine 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 enumeration type to a subclass of UML **Components**. 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 we can be sure usages of components are properly modeled using the **ComponentUsage** relation from our extended metamodel. Now we can define the following constraint to ensure a component does not depend on a less stable component 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)))
```

The impact of the metamodel extension described above on the model of the concrete architecture is quite clear: Each component and interface must be represented by one of the new subclasses we introduced instead of **Component** and **Interface** from the original UML metamodel. The association refinements and the constraints make sure each component provides only interfaces of appropriate types and does 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.

If we take a look at the interfaces provided by the **FrontController** component, we notice that it provides a view interface which offers some frequently used pages as login or error screens. This is obviously a violation of 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. Nevertheless this

approach may be improved regarding the prohibition of the use of elements from the original metamodel which have become unnecessary. The most serious problem caused by the extension of the metamodel 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 we want to discuss 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 and tools the Meta Object Facility [21] (MOF) has been introduced by the OMG to describe models of metadata in a format independent of platform and manufacturer. A limited number of tools supporting the specification of executable models using MOF [1] or the simpler but less expressive EMOF [7] is available. In this section, the adequacy of MOF for the specification of DSLs will be discussed.

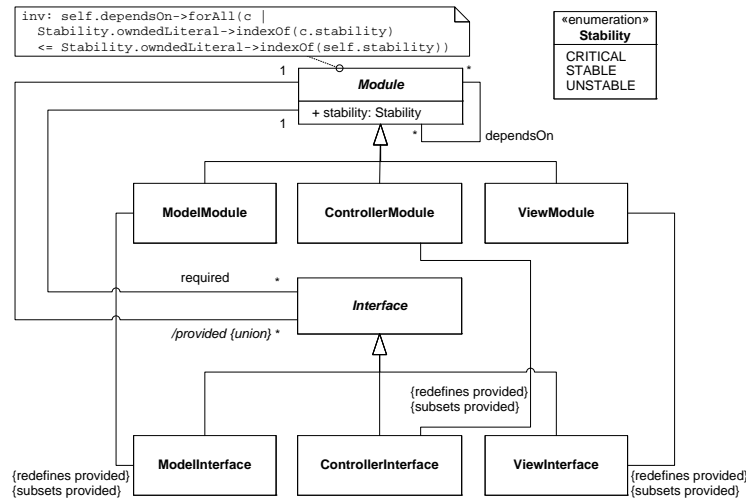


Fig. 6. A MOF model for the MVC design pattern

Figure 6 shows the MOF specification of a simple metamodel to distinguish between model, view and controller components or interfaces. It is basically 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 declared as abstract classes, which keeps us from having to define a constraint to ensure they cannot be instantiated (cf. section 4). Third, the `provided` property of the `Module` class has changed. It is now the derived union of its subsets (which may be determined by some different derivation rule in turn), instead of the derivation rule taken over from the UML specification into our extended metamodel.

The metamodel described in MOF defines an abstract syntax for models of software architectures designed according to the MVC pattern. A concrete syntax is not introduced, so an excerpt from the concrete architecture of our sample application is shown as an object diagram in figure 7. To be able to build a graphical editor for DSLs defined in MOF, a concrete syntax needs to be defined. Frameworks that support the building of editors and tools based on MOF or EMOF models are currently being developed by several organisations [17, 9].

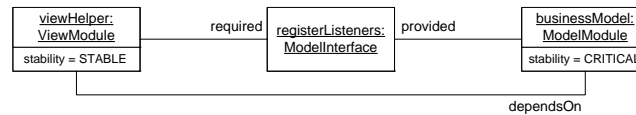


Fig. 7. An excerpt from the architecture model of the Pet Store

The figure shows the coupling of the model and the view subsystem, which is realized by the `registerListeners` 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

```

inv: self.dependsOn->forall(c |
    Stability.ownedLiteral->indexOf(c.stability)
    <= Stability.ownedLiteral->indexOf(self.stability))
  
```

is fulfilled in the example, so we have a valid model here.

As we have seen, the most important advantage of building a new metamodel is the almost unlimited possibility to adopt the metamodel to our needs and preferences. The association concepts of MOF provide a much more precise and convenient way to specify relationships between the introduced metaclasses. Besides that, the elements can be defined in a way that makes the specification of additional constraints in OCL easier 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, especially in areas of application where UML is already used as a standard modeling language, 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

The definition of domain specific languages is addressed by a series of publications, many of which deal either with lightweight or heavyweight extensions of UML or with the specification of new languages by means of MOF.

Berner et al. discuss the use of profiles to extend the UML 1.1 to user specific needs [3]. They distinguish between four classes of stereotypes, which differ from each other in terms of expressive power, ease of proper use and impact of abuse on the customized language. They also admit, that the greater the expressive power of a stereotype is, the more dangerous and difficult it becomes to use in general.

Dong and Yang suggest to use UML profiles for the description of architectural styles in [6], but their approach is focused 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 [12] investigates the differences between stereotypes in UML 1.x and UML 2.0 and points out some issues about their specification from a set theoretical point of view. However, there are some serious flaws in this publication. For instance, the authors conclude 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 [10] 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 is also a number of commercial, proprietary meta-case-tools available (see [13] for an overview), 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.

	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

8 Future Work

From the benefits and drawbacks of the approaches discussed in the previous sections follows the necessity to develop an approach which allows to combine the advantages of UML profiles and metamodeling. 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 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:

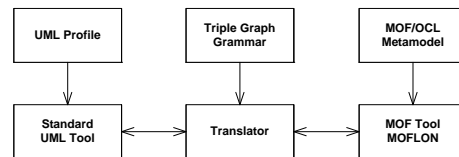


Fig. 8. Combination of UML profiles and metamodel based technologies

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 [16, 27] 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, 14], 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. Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 249–264. Springer, 1999.
4. B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986.
5. 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.
6. J. Dong and S. Yang. Visualizing design patterns with a uml profile, 2003.
7. Eclipse modeling – emf – home, 2007. <http://www.eclipse.org/modeling/emf/>.
8. 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.
9. Graphical Modeling Framework, 2007. <http://www.eclipse.org/modeling/gmf/>.
10. 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.
11. R. Heckel, M. Lohmann, and S. Thöne. Towards a UML Profile for Service-Oriented Architectures. In *Proceedings of Workshop on Model Driven Architecture: Foundations and Applications (MDAFA)*, *CTIT Technical Report TR-CTIT-03-27*. University of Twente, Enschede, The Netherlands, Jun 2003.
12. 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.
13. Hosein Isazadeh and David Alex Lamb. CASE Environments and MetaCASE Tools. Technical report, Queen's University School of Computing, 1997.
14. Jan Juerjens. *Secure Systems Development with UML*. SpringerVerlag, 2003.

15. Mohamed Mancona Kandé and Alfred Strohmeier. Towards a uml profile for software architecture descriptions. In *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, pages 513–527, 2000.
16. A. Königs. *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Technische Universität Darmstadt, 2007. To appear.
17. 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.
18. 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>.
19. 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>.
20. Object Management Group, Inc. Business Process Modeling Notation Specification, Feb 2006. <http://www.omg.org/cgi-bin/apps/doc?dtc/06-02-01.pdf>.
21. 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>.
22. Object Management Group, Inc. Object Constraint Language, May 2006. <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>.
23. Object Management Group, Inc. Unified Modeling Language: Infrastructure, Feb 2007. <http://www.omg.org/cgi-bin/apps/doc?formal/07-02-06.pdf>.
24. Object Management Group, Inc. Unified Modeling Language: Superstructure, Feb 2007. <http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf>.
25. Rational Software. *Rational Unified Process – Best Practices for Software Development Teams*, 1998. http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf.
26. W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
27. 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.
28. Petri Selonen and Jianli Xu. Validating uml models against architectural profiles. *SIGSOFT Softw. Eng. Notes*, 28(5):58–67, 2003.
29. Simulink - simulation and model-based design, 1994–2007. <http://www.mathworks.com/products/simulink/>.
30. Sparx Systems Pty Ltd. Enterprise Architect – UML Design Tools and UML CASE tools for software development, 2000–2007. <http://www.sparxsystems.com.au/products/ea.html>.
31. Stephen Stelting and Olav Massen. *Applied Java Patternes*, chapter 4, pages 208–219. Sun Microsystems, Inc, 2002.
32. Sun Microsystems, Inc. Java pet store architectural overview, 2001. <http://java.sun.com/blueprints/code/jps11/archoverview.html>.
33. Sun Microsystems, Inc. Java ee at a glance, Jun 2006. <http://java.sun.com/javaee/>.