Beyond Stereotyping: Metamodeling Approaches for the UML

Ansgar Schleicher, Bernhard Westfechtel
Department of Computer Science III
Aachen University of Technology
Ahornstr. 55, 52074 Aachen
(schleich|bernhard)@i3.informatik.rwth-aachen.de

Abstract

The UML is currently being used as the universal technique for modeling object-oriented applications across a wide range of domains. Developing a truly adequate uniform modeling technique in the face of these diverse domains seems an unsolvable quest and contrasts domain specific software engineering activities.
Recently, many adaptations to the UML have been made to reflect a domain's world view. These adaptations often exceed the UML's own extension mechanisms and result in yet another urban UML slang.
However, domain-specifically adapting the UML metamodel becomes increasingly important in the context of model checking and code generation mechanisms. Therefore solutions should be found to fully support metamodeling within the UML and UML CASE tools.
The paper discusses and evaluates the UML's inherent as well as proprietary metamodeling approaches and will provide domain driven ideas for a meta-modeling approach for a diversly used Unified Modeling Language

1. Introduction

After a wide variety of object-oriented modeling languages was created particularly in the 90‘s, the *UML* [3] was introduced as a standard notation in order to overcome the upcoming confusion. To make it a *general-purpose modeling language* usable in a rich spectrum of application domains, the designers of the UML decided to include a comprehensive set of modeling techniques for analysis and design as well as structural and behavioral modeling. In this way, they hoped to offer UML users all support they require for their specific applications.

However, it was recognized early that it is difficult to develop a single modeling language suiting the needs of different application domains. This seems to contrast with the goals of *domain-specific software engineering* which is concerned with the design of modeling languages that adequately support the concepts of a specific domain.

As a compromise between the requirements for a standard notation and for domain-specific modeling, the UML was designed as an *extendible modeling language*. In this way, the users of the UML would be able to tailor the language to their specific requirements by introducing domain-specific model elements. On the other hand, these extensions would be performed in a way that conforms with the UML standard.

In this paper, we compare different approaches to extending the UML. We are interested in how the UML may be extended such that

- the extensions are easy to understand (*readability*),
- the semantics of domain-specific concepts may be expressed (*expressive power*),
- the extensions may be made restrictive (*restrictive power*),
- domain-specific constraints may be easily checked (*checkability*), and
- the extensions still conform to the UML, i.e., they must not redefine UML model elements in arbitrary ways or define completely new UML elements (*conformance*).

Clearly, the requirements to extension mechanisms depend on the respective application. In this paper, we study applications that require domain-specific models with well-defined semantics. This is crucial when models are required to be executable or code has to be generated from a model. *Semantic domain modeling* puts high demands particularly on expressive and restrictive power as well as on checkability. In addition, readability and conformance are general requirements that have to be addressed anyway.

The rest of this paper is structured as follows: In Section 2, we introduce a case study (workflow modeling) that is used in Section 3 as a running example to present and compare different approaches to extending the UML.

Finally, Section 4 summarizes the comparison and draws some conclusions.

2. A case study: workflow modeling

Below, we introduce a case study that will be used for discussing various ways of extending the UML in the rest of this paper. The case study is drawn from the area of workflow management for development processes in engineering. In particular, we have studied *software engineering processes*; the example presented below is taken from this domain. The details of modeling software engineering processes do not matter very much here; the interested reader may instead refer to [8].

We have chosen this example for multiple reasons. It is realistic and studied by a variety of research groups (e.g. [6]). It is out of the scope of mainstream UML applications and thus not supported within the UML meta model or available UML CASE tools. In addition, it falls into a group of application domains for the UML that are dependent on rigorous model checking and code generation capabilities, because workflow models are usually simulated or executed within a distributed environment.

A *workflow management system* (WfMS) is a system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications [10]. The *process definition* defines the steps (activities) to be executed as well as their control flow and data flow relationships.

Workflow management systems have been successfully applied for routine processes e.g. within office automation applications. In contrast to these, development processes are highly creative and cannot be planned fully in advance. The clear separation between planning (build) and execution (run) of a workflow, as implemented in classical WfMSs, cannot be upheld in this context. Our approach, called *dynamic task nets [7]*, takes this challenge into account and allows for the interleaved planning, execution, analysis and monitoring of a workflow.

We will only roughly sketch the functionality of our WfMS in this paper by looking into a process for handling change requests of a software system as presented in Figure 1. Each box denotes a *task*, the execution state of which is represented by an icon. Solid thin lines stand for *control flows* which determine the order of task execution. Control flows are refined by *data flows* (dashed lines), which are only shown between two tasks (Change Module B and Test B). Data flows connect *output* and *input parameters* of tasks (black and white circles, respectively). Finally, *feedback flows* (solid thick lines) indicate cycles within the development process.

Within the sample process a redesign of the application has been performed after the change request has been analyzed. According to the new design changes have to be applied to modules B and D (hatched filling) and a new module C has to be implemented. At this time new tasks for changing or implementing these modules and for bottom-up testing of the changed system parts are created within the task net. Discovered errors during the test can be reported back to the responsible task through a feedback relationship which causes another replanning step

Looking at this process it becomes obvious that we are dealing with continuously evolving structures of interrelated process objects.

This is where the UML plays an important role. So far, we have given an example of a software process instance represented by a dynamic task net that is maintained by the workflow engine. To drive the workflow engine, a process definition is required, i.e., the processes to be supported have to be modeled. Why not use the UML for this purpose? In particular, the above sketched *process evolution* can be modeled in a very natural way using an *object-oriented approach*: Tasks are represented by objects which are dynamically created, connected by flow relationships, executed, etc. Furthermore, using a standard notation makes it easier to define and communicate process models.

We apply the UML to software process modeling by using class diagrams for structural modeling. Within class

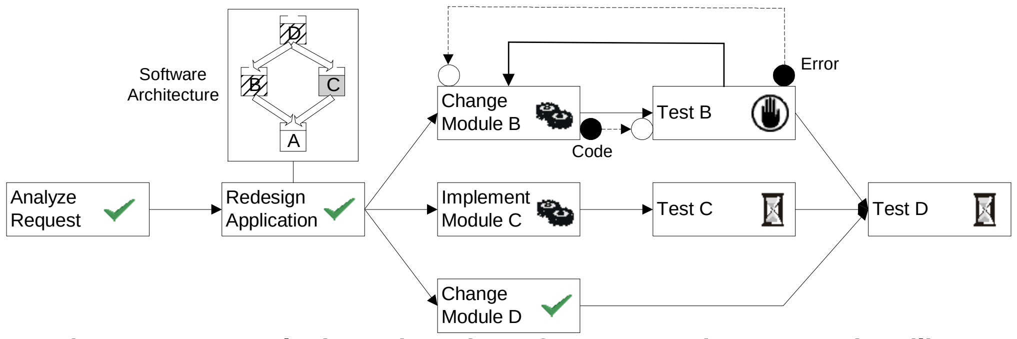


Figure 1 – A sample dynamic task net for an extension request handling process

diagrams classes of tasks and control, data and feedback flow associations are modeled which constrain the evolution of instance-level task nets as shown in Figure 1. State and collaboration diagrams are used for behavioral modeling which will not be discussed in the following [8]. A UML model is transformed into a an executable model intrepretable by the workflow engine [15].

As an example, Figure 2 displays a *class diagram* for the sample change request process introduced above. The screenshot was taken from Rational Rose, which we have employed as a process modeling tool (further details will be discussed in the next section). In the class diagram, we have used *stereotypes* to distinguish between different kinds of model elements such as task classes (stereotype <<Task>>), input and output parameter classes (white and black circles, respectively), and associations for control flows (<<cflow>>), data flows (<<dflow>>), and feedback flows (<<fback>>).

The class diagram states that in the general case a change request process is composed of exactly one Analyze Request and exactly one Redesign task, respectively. The latter is followed by an arbitrary number of Implement Module and Change Module tasks. With respect to testing, the class diagram does not distinguish between tests for changed and new modules. The reflective control flow association serves to express the bottom-up order of testing. Feedback from testing to implementation is represented by a feedback flow association (please note that the class diagram contains only a single example of a feedback association). The dynamic task net in Figure 1 is a valid instance of this class diagram.

It is important that this kind of application requires not only extensions, but also restrictions of the UML. The extensions are used to offer the elements of the underlying process metamodel to the user. The user has to stick to the underlying metamodel (dynamic task nets); otherwise, it is impossible to generate code for driving the workflow engine. Thus, using the UML as a *modeling frontend* to a WfMS puts high demands on expressive and restrictive power as well as on checkability.

3. Metamodeling approaches

The *UML* is frequently used for software and component development and database schema design [13]. As a consequence, UML CASE tools often ship with C++, Java and IDL *code generators* and *schema generators* for common database management systems.

However, the UML becomes increasingly important as a modeling language across *various domains* like multimedia application design, mechanical engineering or workflow modeling [6] as in our case. Since neither the UML nor UML-based CASE tools can incorporate adequate support for every possible development domain, *metamodeling facilities* are of great importance.

A domain-specific metamodel serves as a formal definition of an extension to the UML for the modeling domain. It adds *more semantic depth* to the standard metamodel and thus builds a *foundation* for *model analysis* and *code generation*. Providing good metamodeling sup-

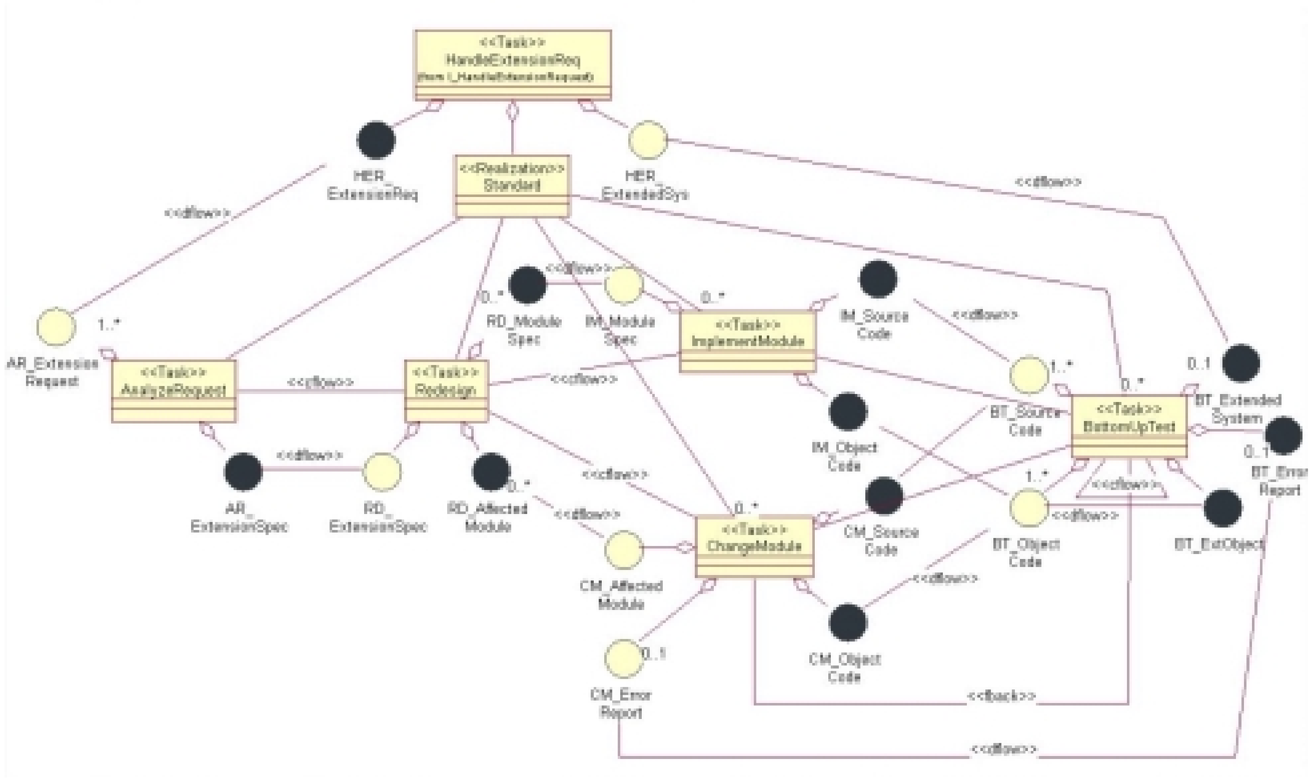


Figure 2 – A sample class diagram for the extension request process

port within the UML enables solution developers to seamlessly integrate new extensions into CASE tools and build special-purpose solutions outside of the standard UML applications.

However, the support for metamodeling within the UML is weak and while many CASE tools are open to the implementation of manifold extensions they do not have an inherent metamodeling and corresponding model checking support.

Driven by our approach to workflow modeling in the UML the following subsections present four options for metamodeling and corresponding tool support. Section 3.2 presents a pragmatic implementation-driven approach to model checking and code generation with respect to a given metamodel. An approach for metamodeling relying on the UML's inherent extension mechanisms is presented in Section 3.3. Sections 3.5 and 3.6 discuss two alternative methods for controlled extensions of the UML metamodel itself.

The evaluation of an approach is based on the following aspects:

- *Readability:* A metamodel should be human readable and understandable as it defines the syntax and semantics of a modeling language. Readability enhances modeling comfort and acceptance for a novel UML application.
- *Expressive Power*: All metamodel aspects should be expressable with the approach.
- *Restrictive Power*: A domain-specific metamodel should not only extend the UML metamodel but also restrict it to meaningful structures with respect to the domain (e.g. it makes no sense to allow multiple inheritance in a design model's class diagram if Java-code is to be generated from the model).
- *Checkability*: Tools should be able to check a model for consistency with a given metamodel. This is especially important, if proprietary code generators are to be implemented. Therefore the metamodel must have well-defined semantics.
- *Changeability*: Domains and thus metamodels evolve continuously. Changes should be performed in a controlled way and have local impact only.
- *Conformance*: A metamodeling approach should not allow for arbitrary UML extensions which result in UML dialects hardly supportable by tools. Within this paper we define conformance as follows: A domain-specific metamodel SM conforms to the UML metamodel UM, if a valid instance of SM is a valid instance of UM after the domain-specific metaclasses of every model element in the instance of SM are replaced by the corresponding metaclass in UM that they extend. Inherent to this definition are the following prerequisites for every domain-specific metamodel: New metaclasses have to have a (transitive) superclass within the UML metamodel and instances of the new metaclasses have to be substitutable for instances of the original metaclass.

An approach fulfilling all of these issues goes far beyond the metamodeling facilities included within the UML (cf. Section 3.3). Especially the aspects concerning readability, restrictive power and checkability are totally neglected. Because of the rather mediocre metamodeling facilities within the UML, a definition of conformance was obsolete. If *full-fledged metamodeling support* is to be integrated into the UML an appropriate definition of *conformance is very essential*.

3.1. Classification of stereotypes

The UML contains three extension-mechanisms: *stereotypes*, *tagged values*, and *constraints*, of which the first two are supported by the market-leading UML CASE tool *Rose*. In [2] a classification schema for the use of stereotypes as a metamodeling facility is presented. Four different kinds of stereotypes are distinguished which will be described in the following.

Decorative stereotypes are pure manipulations of the concrete syntax and are used to replace a symbol of a model element. They are used to adapt the notation of UML to a specific domain. The black circles of Figure 2 are decorative stereotypes if no further restrictions are formulated on their usage. In that case they remain regular UML classes.

Descriptive stereotypes introduce new pragmatic elements that do not change the semantics of the UML. Descriptive Stereotypes are a secondary classification of a valid UML metamodel element. If we call the black circles from Figure 2 *output parameter classes* we provide a new pragmatic element in the context of workflow modeling. However, within a UML model a class stereotyped with a descriptive stereotype remains an instance of the original UML metamodel element. No constraints regarding the syntax or semantics are added to the original metamodel element by providing a set of descriptive stereotypes for it.

Restrictive stereotypes are new semantic elements added to the UML. They are first class members of the new language and include a formal definition of syntactical and semantical constraints regarding their usage within a model. However, they do not *change* the base language and its semantics, they can only *extend* it. The constraints of the stereotyped metamodel element apply to the newly introduced metamodel element as well. An instance of a restrictive stereotype remains a valid instance of the stereotyped original metamodel element.

Redefining stereotypes provide the means to replace any given metamodel element through a new one and defining a completely different set of constraints for it.

This induces radical changes in the original language and results in a new modeling language being defined.

Within this and the following subsections descriptive and restrictive stereotypes will be discussed. Decorative stereotypes are omitted from the discussion because they lack any kind of metamodeling support; redefining stereotypes are omitted because they alter the UML and create a UML dialect which can by default not be supported by a CASE tool. The use of redefining stereotypes violates conformance.

3.2. Descriptive stereotypes

Since CASE tools like Rose do not support anything more sophisticated than descriptive stereotypes, our first approach to incorporate the metamodel of dynamic task nets into the UML (and into Rose) was to map metamodel classes to descriptive stereotypes. This is a rather simple task: Every metaclass of the modeling domain, like task, input and output parameter, controlflow association etc. is represented by its own stereotype. Some of these stereotypes are symbolized by an own graphical symbol which enhances readability of conforming models.

This approach leeds to a graphically and pragmatically enhanced tool, a diagram of which was presented in Figure 2. The model structure is kept in a stereotyped *package hierarchy* and a process definition is modeled structurally within a *stereotyped class diagram*.

However, the enhanced tool does not support the metamodel by forbidding meaningless structures with respect to the metamodel or the use of unsupported metaclasses of the UML. The process modeler can freely enter any kind of valid UML class diagram. Bearing this in mind, it is impossible to generate a formal, interpretable process definition from the UML model without providing model checking support as well.

We overcame these deficits by *handcoding* a *model checker* and *code generator* via the OLE Automation Interface provided by Rose. Code generation can only start if model checking returned successfully and can thus rely on a consistent model with respect to the metamodel of dynamic task nets.

With respect to our evaluation criterea we can state the following: On the *negative* side we find the *readability*, *changeability*, and *checkability* characteristics of this metamodel-programming approach based on descriptive stereotypes. Since the metamodel is hardcoded into the model checker, it is hard to understand and requires programming skills to introduce changes. Checkability is inherently supported, since the model checking algorithm represents the metamodel. However, the provision of model checking means a lot of work which has to be repeated for every metamodel.

Positive characteristics of this approach are its *expressive* and *restrictive power*. The expressive power is only limited by the capabilites of the underlying CASE tool, since any syntactical or semantical constraint can be hardcoded into the model checker. These constraints are only *checked on demand*. A more suitable approach to checkability would be the realization of *eager checks*, e.g. in the form of a syntax directed editor. The approach is restrictive, since model checking can reject the use of any UML metaclass. *Conformance* is *supported by default*, because in the context of Rose descriptive stereotypes can only be defined for a given UML metaclass. Thus every stereotyped model element remains an instance of a valid UML metaclass.

The use of descriptive stereotypes is widely spread. Especially in contexts, where *a new modeling methodology* is to be introduced, graphical and pragmatic enhancements play the key role. Model checking and code generation are of no primary interest in these cases. [1] and [9] introduce modeling methodologies for hypermedia application design and real-time systems modeling in this fashion. Another application area for descriptive stereotypes stems from the mapping of formally defined proprietary modeling languages onto the UML. In these cases, the formal metamodel lies outside of the tool [16].

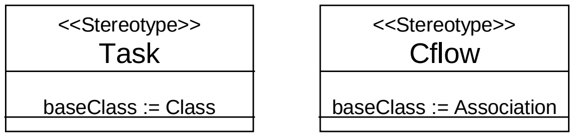
3.3. Restrictive stereotypes

Handcoding the metamodel in a given programming language can definetely not be a suitable approach to metamodeling, although it is the only alternative regarding the capabilities of today's CASE tools. However, the UML itself provides more expressive metamodeling techniques than descriptive stereotypes. For every stereotype a set of tagged values can be defined which can be interpreted as metalevel attributes. Additionally, constraints can be formulated on a stereotype (with e.g. the OCL [17]) and its context with respect to the stereotyped metaclass. In this fashion valid structures can be defined.

If a stereotype is constrained it is called restrictive because it may not be placed within every valid UML context. Rather, the associations and (stereotyped) instances of other metaclasses within the context of the stereotyped element will be checked for validity. Additionally, predefined values for metalevel attributes can be fixed through constraints.

Assuming available CASE tool support, using restrictive stereotypes has the inherent advantage of not having to code model checking for every metamodel separately. Rather, the *constraints* can be *specified declaratively* and *checked automatically* by the tool.

Figure 3 contains a small cutout of a metamodel for dynamic task nets based on restrictive stereotypes. In analogy to descriptive stereotyping we need a new stereotype for every domain-specific metaclass which is shown for tasks and controlflow associations at the top of the figure. For every stereotype a set of constraints is



```
Cflow
      -- binary assocation
self.connection.size = 2
      -- cflow associates two elements of type Task
self.connection->forall(ae: AssocationEnd | ae.type.oclisTypeOf(Task))
      -- Association has no aggregation ends and is navigable from both
      -- ends
self.connection->forall(ae: AssociationEnd |
                          ae.aggregation=none and ae.isNavigable = true)
      -- One end depicts the source role of the association
self.connection->exists(ae: AssociationEnd | ae.Name="src")
      -- Another end depicts the target role of the association
self.connection->exists(ae: AssociationEnd | ae.Name="trg")

Task
      -- Only one controlflow association between a pair of tasks
self.associationEnd->select(ae | ae.Name="src").association.associationEnd
      ->select(ae | ae.Name="trg").type->asBag->forall(t1, t2 | not t1=t2)
```

Figure 3 – Definition of restrictive stereotypes

defined restricting the use of this stereotype to meaningful structures with respect to dynamic task nets. In this case, the following restrictions are expressed: A controlflow association (stereotype Cflow) is a binary, directed and bidirectionally navigable association with no aggregation ends. It connects two elements of stereotype Task. Between every pair of tasks there may only be one controlflow association with the same direction. The corresponding constraints are formulated on the stereotypes Cflow and Task, respectively.

The metamodeling approach using restrictive stereotypes *lacks readability* even though the gain over hand-coded constraints is obvious.

The *expressive power* of OCL-constraints is *sufficient* for most purposes even though not all constraints of the UML metamodel itself could be properly expressed with the OCL [12]. In contrast, the *restrictive power* of this approach is *not sufficient*. Restrictive stereotypes are *pure extensions* to the UML metamodel which means model checkers based on constraint interpretation would not reject the use of any valid UML construct within a model. However, with stereotyped namespaces (e.g. packages) and defined constraints for each namespace restricting the valid stereotypes for contained elements, restrictive power might be adequate for some cases. For each package-stereotype the stereotypes of contained elements are precisely defined. A task package (<<TaskP>>) may contain other task packages, interface and realization packages (<<InterfaceP>>, <<RealizationP>>); an interface package may contain task and parameter classes as well as some stereotyped associations and so forth.

The *checkability* of constraints is currently *limited*, since the OCL does not have defined execution semantics (although this topic is adressed by the research community, [14]). Providing these semantics and implementations within CASE tools would lead to full checkability of restrictive stereotypes.

Since constraints are locally defined for one stereotype *changeability* is *good*. Metamodels expressed using the restrictive stereotype approach *conform to the UML*, because only predefined extension mechanisms of the UML are used.

Restrictive stereotypes are used in cases where rigorous semantics have to be defined for modeling elements. The predefined extensions for software development and business modeling presented in the UML specification document [12] are metamodels defined through restrictive stereotypes, although restrictions in these cases are defined through tables rather than OCL constraints. Within these tables the source and target types of stereotyped associations are constrained. However, these tables can easily be translated into OCL constraints which would enable the definition of valid cardinalities for these associations, too.

3.4. Classification of metamodel extensions

Due to the restrictions of the UML's inherent metamodeling mechanisms, especially concerning modeling comfort, numerous domain-specific metamodels have been defined by *extending* the *UML's* own *metamodel* with new metaclasses and meta-associations. The advantage of this approach is the possibility to use class diagrams for the definition of many structural constraints rather than OCL.

Extending the UML's metamodel means adding new metaclasses and meta-associations to it. Since the UML metamodel itself is a valid instance of the MOF meta-metamodel, extending the UML metamodel means defining a new modeling language by instantiating a new MOF model.

We have to distinguish between two cases of such metamodel changes: In the first case arbitrary metaclasses can be implanted into the original metamodel, regardless of their superclass. Meta-associations may be defined between any set of metaclasses regardless whether they are of a refining nature or completely new to the metamodel. In the second and more restrictive case, new metaclasses are only valid if they have a superclass within the original UML metamodel. Meta-associations may only be introduced if they refine a meta-association from the original metamodel. In the first case, which we will call *uncontrolled* in the following, a UML dialect is created, where the semantics and notation of new elements can not be interpreted by any CASE tool. In the latter case, which we will call *controlled*, an instance of a newly created metaclass can be substituted for an instance of an original metaclass. The new metaclass thus provides at least the semantics of its original superclass and can thus be handled as such by any UML CASE tool.

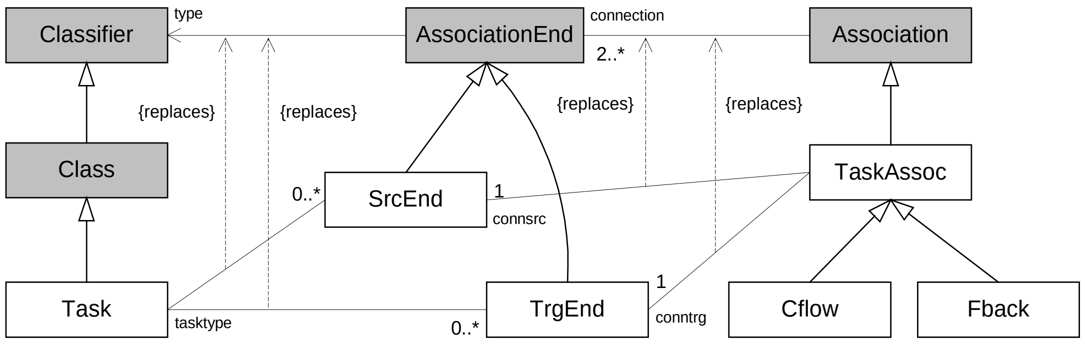
In the following section we present *regular metamodel extensions* as a means to define a domain metamodel. We call these extensions regular, as they start out with the existent UML metamodel and define extensions to it. A different approach is taken by *restrictive metamodel extensions* which imply the full restrictive power needed by application domains like workflow modeling. In both cases we will shortly discuss the uncontrolled case, even though it violates conformance in both cases and is unsupportable in UML CASE tools.

3.5. Regular metamodel extensions

As an example the cutout from the dynamic task nets' metamodel concerned with controlflow associations as presented in Section 3.3 formulated through controlled metamodel extensions is shown in Figure 4 (UML metaclasses are shaded gray). Some new metaclasses are introduced: A metaclass Task inheriting from Class, metaclasses TaskAssoc, Cflow and Fback inheriting from Association and two new metaclasses defining specialized AssociationEnds namely source and target ends for a directed, navigable association.

Introducing a new metaclass as a subclass to an original UML metaclass is equivalent to defining a stereotype. The benefit lies in the ability to use regular class diagrams to define structural constraints on the new model elements.

In the example we define meta-associations that allow for the connection of one task association to exactly one source association end and one target association end. In addition, we specify that a task class may be connected by an arbitrary number of source and target association ends. These new meta-associations implicitly *refine original meta-associations* (controlled case). Specifying the meta-associations in this way is superfluous, since the new metaclasses inherit these meta-associations from their respective UML metaclass. Thus, the newly introduced meta-associations between the new metaclasses do not define any structural restrictions with respect to the embedding of their instances into a model. Only if dependencies between the original meta-associations and their



```
SrcEnd                         Task
self.Name = "src"              self.SrcEnd.TaskAssoc.conntrg.tasktype->asBag
self.aggregation = none                          ->forall(t1, t2 | not t1 = t2)
self.isNavigable = true
```

Figure 4 – Example for metamodel extensions

replacements are defined together with a replaces-constraint, an effective restriction is established. In this case e.g. a task class may not be connected by any association end other than the new source and target ends.

Using UML metamodel extensions to implant a domain-specific metamodel into the UML still does not free us from *specifying constraints*. For example the metaclasses SrcEnd and TrgEnd have to be further specified by predefining some of their attribute values. The constraint denoting the uniqueness of a controlflow association between two tasks still has to be formulated in OCL as well. However, with the extended metamodel it is easier to formulate and read.

Metamodel extensions *enhance readability* of the model and the OCL constraints. Using class diagrams to define structural constraints is the natural choice in the context of the UML and it provides a modeler with a notation he is used to.

In the controlled case *expressive power* is *equivalent* to *restrictive stereotypes*. The uncontrolled case even exceeds the expressive power of redefining stereotypes, because arbitrary metaclasses and meta-associations between metaclasses may be defined. Its expressive power is unrestricted but the result is a modeling language out of the scope of UML.

Restrictive power is *equivalent* to *restrictive stereotypes*. All of the structural constraints included in class diagrams could be formulated in OCL just as well. Unfortunately, this means that regular UML model elements cannot be prohibited within a model.

Changeability is on the *same level* as with any rigorously defined *domain-model* written in UML, since the same language constructs are used on metamodeling and modeling level.

Conformance to the UML metamodel is *only given* in the *controlled case*, where instances of every newly introduced metaclass and meta-association can be handled as if they were instances of their respective UML metaclass. Of course checkability is lost if a tool cannot interpret and support the metamodel.

Metamodel extensions are generally used in both the controlled and the uncontrolled fashion. [11] describes a metamodel extension to support reuse and evolution of model components through reuse contracts. This metamodel extension is controlled, since all introduced meta-associations refine existing meta-associations from the UML metamodel. However, these dependencies are not explicitly included in the extended metamodel. In contrast, [13] deals with the use of class diagrams for object-oriented database design. Metamodel extensions are necessary to allow for the specification of integrity constraints. The presented metamodel extensions are uncontrolled, because meta-associations exist that are not refinements of valid UML meta-associations.

An approach to use the UML's package concept to define multiple metamodel extensions is described in [4]. It does not include a discussion of whether controlled or uncontrolled extensions are to be supported as it focusses on the addition and redefinition of attributes in submetaclasses and the joining of two metaclasses in cases of multiple package inheritance.

3.6. Restrictive metamodel extensions

The *weakness* of the approaches discussed so far is that automatic *model checking* can only be done *locally*. This means that it can be checked whether an instance of a domain-specific metaclass is used correctly within the model but it cannot be checked whether the complete model is consistent with the domain's metamodel, because it may contain arbitrary model elements of the UML. In the context of code generation the most *important requirement* is the *consistency* of a model regarding a given domain-specific metamodel. This consistency is usually reached by providing model checking support within the modeling tool. In the context of the UML automatic model checking could be supported by CASE tools, if a metamodeling approach were used that most of all allows to formulate restrictions on the original UML metamodel.

In this section we introduce a metamodeling approach based on metamodel extensions as presented in Section 3.5 but *enhances* its *restrictive power* to meet code generation requirements. The approach consists of three main ideas:

1. Make the UML metamodel consist of *abstract* and thus non-instantiable *metaclasses* only.
2. Define instantiable metamodels on the basis of this abstract metamodel by using *generalization* relationships between packages.
3. Make it *obligatory* for every instantiable metaclass to have an *abstract superclass* from the UML metamodel and every newly introduced association to be a refinement of an original association.

Figure 5 shows how the common UML metamodel and other domain-specific metamodels can then coexist in different packages. In the context of our workflow management activities we defined additional metamodels for product and resource management. Each of these metamodels is independent of the others and can be used separately. However, in order to provide highly integrated workflow management the three metamodels can be integrated. For this purpose we define a fourth package for integration aspects. This package is derived from the three afore mentioned metamodel packages.

Adequate tool support could then allow for the *selection* of current *metamodel packages* and – on the basis of

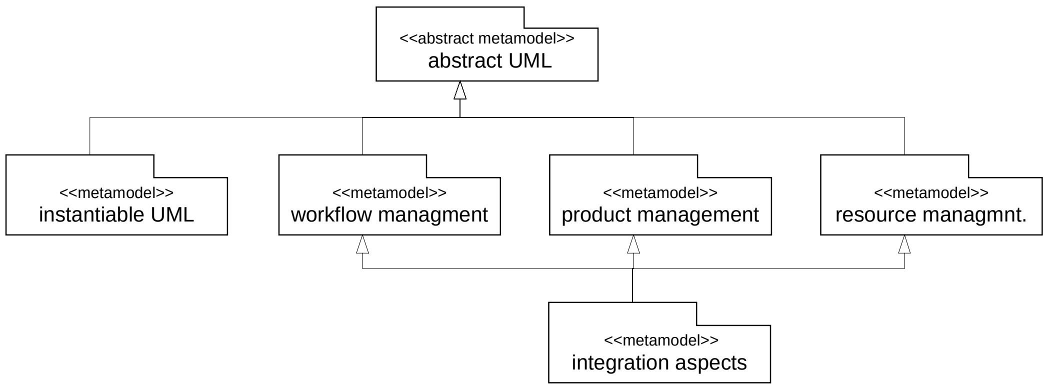


Figure 5 – Metamodel packages

interpreters for class diagrams and OCL-constraints – provide automatic model checking.

This metamodeling methodology has many inherent advantages: *Readability* is *good,* since metamodels are well separated and self-contained and their interdependencies are visible at first glance in diagrams as presented in Figure 5. Class diagrams can be used as a mechanism to define structural constraints. These features *enhance changeability* as well.

The *expressive power* is *equivalent* to *regular metamodel extensions* as described in subsection 3.5 but *restrictive power* is *much greater*. From the presented approaches only this one allows for the rejection of standard UML classes, which provides the optimal basis for automatic model checking. With this approach models can be checked globally for consistency with the current metamodel. *Conformance* with the UML *is enforced*, since every metaclass in every metamodel has to be a specialization of an original abstract metaclass and the introduction of associations into metamodels is limited according to the definition of conformance.

4. Conclusion

We have presented and compared different metamodeling approaches for the UML. To this end, we have used a case study from software process modeling in which the UML is employed for defining process models from which executable code may be generated for driving a workflow engine. More generally, we have investigated a class of UML applications which is characterized by domain modeling and code generation. Domain modeling demands for metamodel extensions, while code generation in addition requires the enforcement of restrictions.

For this application class, we evaluate the alternatives presented in the previous section as follows:

- *Descriptive stereotypes* serve to express the elements of the underlying domain metamodel. With the help of descriptive stereotypes, the user may create UML models using the elements of the domain-specific metamodel. Basically, the user is supported at a pragmatic level through the use of icons, colors, etc. However, the domain-specific metamodel is not defined explicitly; rather, it must be hardcoded into an analysis tool which performs model checking in a batch-like fashion and on demand only.
- *Restrictive stereotypes* go beyond the descriptive ones by attaching constraints to stereotyped model elements. These constraints are defined declaratively instead of being hardcoded. If appropriate tool support is available, they can be checked or even enforced. This would obviate the need for a handcoded model checker. Unfortunately, restrictive stereotypes result in an unreadable metamodel definition, as metamodels are expressed by textual OCL constraints.
- *Regular metamodel extensions* make use of full-blown metamodel support. Metamodels are defined within metaclass diagrams, which may still be supplemented by constraints. In the case of restrictive stereotypes, the metamodeler essentially would draw a metaclass diagram (maybe with paper and pencil) and then encode it with the help of OCL constraints. It is much easier and more natural to base metamodeling on metaclass diagrams, as it has been done in meta-CASE tools for a long time before the UML was introduced. Without further provisions, metamodel extensions only offer additional model elements, but they do not exclude the existing ones.
- *Restrictive metamodel extensions* go one step further. By making the standard UML model elements abstract, their instantiation may be prohibited. A domain-specific metamodel may then introduce instantiable model elements. This alternative is superior to the previous one with respect to its restrictive power.

While the designers of the UML have introduced stereotypes as a means for „poor man´s metamodeling“, there are indeed applications that call for first-class metamodeling. This is not yet supported in the UML (1.3), but there are activities going on working toward that goal. This paper may contribute to these activities by contrasting different metamodeling approaches, but also by proposing *controlled restrictive metamodel extensions*, which is – to the best of our knowledge – an original contribution. The mechanisms we propose for control are:

- All metaclasses must be *specializations* of existing ones (i.e., of metaclasses defined in the UML standard).
- Likewise, extensions must not introduce new meta-associations; rather, we allow only for the *replacement* of already existing meta-associations.
- In the case of restrictive metamodel extensions, existing metaclasses are defined as being *abstract* so that their instantiation can be prohibited.

Unfortunately, current CASE tools such as Rational Rose offer only limited metamodeling support (descriptive stereotypes). This paper serves to reinforce the arguments already given in [5] that UML CASE tools should provide first-class metamodeling.

5. References

[1] Baumeister, H., Koch, N., and Mandel, L.: *Towards a UML Extension for Hypermedia design* in France, R. and Rumpe, B.: Proceedings <<UML>>'99 - The Unified Modeling Language., LNCS 1723, pages 614 - 629, Springer-Verlag, Berlin, Heidelberg, New York (1999)

[2] Berner, S., Glinz, M., and Joos, S.: *A Classification of Stereotypes for Object-Oriented Modeling Languages* in France, R. and Rumpe, B.: Proceedings <<UML>>'99 - The Unified Modeling Language, LNCS 1723, pages 249 - 264, Springer-Verlag, Berlin, Heidelberg, New York (1999)

[3] Booch, G., Jacobson, I., and Rumbaugh, J.: *The Unified Modeling Language User Guide,* Addison-Wesley, Reading, MA (1999)

[4] D'Souza, D., Sane, A., and Birchenough, A.: *First-Class Extensibility for UML - Packaging of Pofiles, Stereotypes, Patterns* in France, R. and Rumpe, B.: Proceedings <<UML>>'99 - The Unified Modeling Language, LNCS 1723, pages 265 - 277, Springer-Verlag , Berlin, Heidelberg, New York (1999)

[5] Dykman, N., Griss, M., and Kessler, R.: *Nine Suggestions for Improving UML Extensibility* in France, R. and Rumpe, B.: Proceedings <<UML>>'99 - The Unified Modeling Language, LNCS 1723, pages 236 - 248, Springer-Verlag, Berlin, Heidelberg, New York (1999)

[6] Franch, X. and Ribó, J. M.: *Using UML for Modelling the Static Part of a Software Process* in France, R. and Rumpe, B.: Proceedings <<UML>>'99 - The Unified Modeling Language, LNCS 1723, pages 292 - 307, Springer-Verlag, Berlin, Heidelberg, New York (1999)

[7] Heimann, P., Joeris, G., Krapp, C.-A., and Westfechtel, B.: *DYNAMITE: Dynamic Task Nets for Software Process Management* in Proceedings 18th Int. Conf. Software Engineering (ICSE 18), pages 331 - 341, Berlin, Germany (1996)

[8] Jäger, D., Schleicher, A., and Westfechtel, B.: *Using UML for Software Process Modeling* in Nierstrasz, O. and Lemoine, M.: Proceedings ESEC/FSE '99, LNCS 1687, pages 91 - 108, Springer-Verlag, Heidelberg (1999)

[9] Lanusse, A., Gérard, S., and Terrier, F.: *Real-Time Modeling with UML: The ACCORD Approach* in Bézivin, J. and Muller, P.-A.: Proceedings <<UML>>'98 - The Unified Modeling Language, LNCS 1618, pages 319 - 335, Springer-Verlag, Berlin, Heidelberg, New York (1998)

[10] Lawrence, P.: *Workflow Handbook*, Wiley, Chichester (1997)

[11] Mens, T., Lucas, C., and Steyart, P.: *Supporting Disciplined Reuse and Evolution of UML Models* in Bézivin, J. and Muller, P.-A.: Proceedings <<UML>>'98 - The Unified Modeling Language, LNCS 1618, pages 378 - 392, Springer-Verlag, Berlin, Heidelberg, New York (1998)

[12] OMG: *OMG Unified Modeling Language Specification*, www.omg.org (2000)

[13] Ou, Y.: *On Using UML Class Diagrams for Object-Oriented Database Design Specification of Integrity Constraints* in Bézivin, J. and Muller, P.-A.: Proceedings <<UML>>'98 - The Unified Modeling Language, LNCS 1618, pages 173 - 188, Springer-Verlag, Berlin, Heidelberg, New York (1998)

[14] Richters, M. and Gogolla, M.: *On Formalizing the UML Object Constraint Language OCL* in Ling, T. W., Ram, S., and Lee, M. L.: Proceedings 17th Int. Conf. Conceptual Modeling (ER'98), LNCS 1507, pages 449 - 464, Springer-Verlag, Berlin, Heidelberg, New York (1998)

[15] Schleicher, A.: *Formalizing UML-based Process Models Using Graph Transformations* in Nagl, M. and Schürr, A.: Proceedings AGTIVE '99, LNCS 1779, pages 341 - 358, Springer-Verlag, Berlin, Heidelberg, New York (2000)

[16] Selic, B.: *Using UML for Modeling Complex Real-Time Systems* in Mueller, F. and Bestavros, A.: Proceedings ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, LNCS 1474, pages 250 - 260, Springer-Verlag, Berlin, Heidelberg, New York (1998)

[17] Warmer, J., and Kleppe, A.: *The Object Constraint Language - Precise Modeling with UML,* Addison Wesley, Reading, Massachusetts (1999)